AFRL-IF-RS-TR-1998-168
Final Technical Report
August 1998

VISUALIZATION AND MODELING FOR INTERACTIVE PLAN DEVELOPMENT AND PLAN STEERING

University of Massachusetts

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. 8136

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

AFRL-IF-RS-TR-1998-168 has been reviewed and is approved for publication.

APPROVED: [illegible]

WAYNE A. BOSCO
Project Engineer

FOR THE DIRECTOR: [illegible]

NORTHRUP FOWLER, III, Technical Advisor
Information Technology Division
Information Directorate

VISUALIZATION AND MODELING FOR INTERACTIVE PLAN DEVELOPMENT AND PLAN STEERING

Paul R. Cohen

Contractor:	University of Massachusetts
Contract Number:	F30602-91-C-0076
Effective Date of Contract:	24 July 1991
Contract Expiration Date:	9 September 1994
Program Code Number:	3E20
Short Title of Work:	Visualization and Modeling for Interactive Plan Development and Plan Steering
Period of Work Covered:	Jul 91 - Sep 94

Principal Investigator:	Paul R. Cohen
Phone:	(413) 454-3638
AFRL Project Engineer:	Wayne Bosco
Phone:	(315) 330-3578

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Wayne Bosco, AFRL/IFTB, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE August 1998	3. REPORT TYPE AND DATES COVERED Final Jul 91 - Sep 94

4. TITLE AND SUBTITLE VISUALIZATION AND MODELING FOR INTERACTIVE PLAN DEVELOPMENT AND PLAN STEERING	5. FUNDING NUMBERS C - F30602-91-C-0076 PE - 62301E PR - H136 TA - 00 WU - 15
6. AUTHOR(S) Paul R. Cohen and David M. Hart	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Massachusetts Computer Science Department, Box 34610 Amherst, MA 01003-4610	8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency, 3701 North Fairfax Drive, Arlington, VA 22203-1714 AFRL/IFTB, 525 Brooks Road, Rome, NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-1998-168
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Wayne Bosco/IFTB/(315) 330-3578	
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for Public Release; Distribution Unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report describes research to develop a mixed-initiative plan execution system in which an interactive "plan steering" agent assists a human user in keeping an executing military plan on track. In the Interactive Plan Steering (IPS) architecture we developed, a network of demons watches the executing plan looking for pathological states that indicate the plan may be failing in some respect. When a pathology is detected, a Plan Steering Agent (PSA) engages the user in a visual dialogue, altering the the user to the pathology and offering advice for steering the plan around it. Experiments show that a human operator working together with the IPS system outperforms either 1) the human working alone or 2) the IPS system operating without the human's help.

A new technique for detecting pathologies during plan execution has been developed under this contract. This technique, called Multi-Stream Dependency Detection (MSDD), examines plan execution traces from multiple sources looking for significant dependencies among actions and effects. MSDD will be useful for plan steering by identifying statistical indicators of pathologies that can be used to trigger demons. It will also prove generally useful for other aspects of planner evaluation, such as identifying unexpected causes of plan failure or unanticipated interaction effects between plan actions.

14. SUBJECT TERMS Interactive Planning, Interactive Plan Steering, Plan Steering Agent, Mixed-Initiative Planning, Embedded Agents, CLIP/CLASP, Multi-Stream Dependency Detection	15. NUMBER OF PAGES 104 16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

Abstract

The objective of our work was to develop a mixed-initiative plan execution system in which an interactive "plan steering" agent assists a human user in keeping an executing military plan on track. In the Interactive Plan Steering (IPS) architecture we developed, a network of demons watches the executing plan looking for pathological states that indicate the plan may be failing in some respect. When a pathology is detected, a Plan Steering Agent (PSA) engages the user in a visual dialogue, alerting the user to the pathology and offering advice for steering the plan around it. Experiments show that a human operator working together with the IPS system outperforms either 1) the human working alone or 2) the IPS system operating without the human's help.

A new technique for detecting pathologies during plan execution has been developed under this contract. This technique, called Multi-Stream Dependency Detection (MSDD), examines plan execution traces from multiple sources looking for significant dependencies among actions and effects. MSDD will be useful for plan steering by identifying statistical indicators of pathologies that can be used to trigger demons. It will also prove generally useful for other aspects of planner evaluation, such as identifying unexpected causes of plan failure or unanticipated interaction effects between plan actions.

The CLIP/CLASP instrumentation and statistical analysis package was integrated into the ARPI's Common Prototyping Environment, making it available to other researchers in the initiative for evaluation of their planning systems. Several workshops were held to introduce CLIP/CLASP to ARPI participants.

In research funded by an AASERT award we are developing a taxonomy of optimal monitoring strategies that intelligent agents will use to predict plan failures in complex, real-time domains. Early warning of plan failure provides valuable lead time for replanning. However, the cost of monitoring must be weighed against the benefits of early failure detection. In previous work we showed that a strategy called interval reduction is often superior to periodic monitoring, the strategy most commonly employed in intelligent agent architectures. We have also developed an efficient, anytime planner that controls the rate of sensing during plan execution.

1 Executive Summary

1.1 Introduction

Research conducted under this ARPA/RL contract has focused on the development of visualization and modeling tools and their use for Interactive Plan Steering (IPS). IPS predicts the occurrence of pathological conditions in a transportation network, alerts a user as these problems arise, and assists the user in steering activity to alleviate the pathological conditions. At the same time we worked under an AASERT award to explore optimal monitoring strategies for agents in embedded environments. In this report we present significant results of these efforts in three areas: 1) implemetation of a prototype IPS system, 2) delivery to the ARPI Common Prototyping Environment (CPE) of an instrumentation and analysis package for planner evaluation, and 3) identification of several monitoring strategies that are superior to those commonly employed by embedded agents.

Interactive Plan Steering. We developed a prototype IPS system for managing the execution of transportation plans. This system assists a user who is managing traffic in a shipping network that has many of the important domain characteristics of the ARPI's IFD2 scenario. To support this effort we took the ARPI's Prototype Feasibility Estimator (PFE) and, using domain knowledge supplied by ISX Corporation, implemented a discrete event simulator of the shipping network we call TRANSSIM. Using TRANSSIM to execute shipping "schedules" (i.e., simple movement requirements that specify departure and arrival ports and deadlines for arrival), we developed an IPS architecture that experiments show performs better than either an unassisted computer program or a human steering the traffic unassisted. We also report on the development of a new algorithm for detecting pathological conditions in network activity called Multi-Stream Dependency Detection (MSDD).

Tools for Evaluating Planners. We provided the CLIP/CLASP system for ARPI researchers interested in empirical evaluation of program behavior. CLIP/CLASP is an online laboratory environment that allows researchers to instrument their programs, run experiments, and analyze the results of those experiments – all in one integrated Common Lisp environment. CLIP/CLASP can be used for performance evaluation and for building models of domain activity (such as bottlenecks at seaports) and program behavior (how much improved is our new bottleneck detection algorithm over the old?). The first version of CLIP/CLASP was integrated into the CPE in 1993, and since that time several upgrades have been installed. We conducted several workshops to introduce the ARPI community to CLIP/CLASP.

Investigating Monitoring Strategies for Embedded Agents. Suspecting that commonly employed monitoring strategies for embedded agents are suboptimal in many cases, we investigated several complementary monitoring problems in order to find optimal strategies. In one case we show that a strategy called *interval reduction* monitoring is superior to the commonly employed periodic monitoring strategy. In related work we developed an efficient, anytime planner that controls the rate of sensing during plan execution, striking a cost-effective balance between open loop systems that don't monitor at all and closed-loop systems that monitor after each action.

Our work in each of these areas is summarized in the next subsection, Summary of Technical Results. Also included in the next section is summary information about: publications (Sections 1.3), conference presentations (Section 1.4), awards (Section 1.5), technology transfer (Section 1.6) and software prototypes (Section 1.7). Full details of these research results are provided in Sections 2-6.

1.2 Summary of Technical Results

1.2.1 Interactive Plan Steering

Under this contract we developed an Interactive Plan Steering (IPS) system for managing the execution of transportation plans. IPS predicts the occurrence of pathological conditions in a transportation network (modeled after the IFD2 scenario), alerts the user as they arise, and assists the user in taking steering actions to avoid the problem. We developed IPS in the TRANSSIM system, a configurable and instrumented simulation testbed.

When a plan involves large numbers of events over time, it can be difficult or impossible to tell whether those events are unfolding "according to plan" and to predict the effects of dynamic plan modifications. Neither task is particularly challenging for small-scale planning problems: the entire state of a plan can be stored, and plan modifications can be evaluated by search. However, for the large-scale transportation planning problems faced by USTRANSCOM, it is both necessary and difficult – technically and scientifically – to assess the current state of a plan and to predict the effects of plan modifications.

In the absence of informed intervention, a plan can evolve into a pathological state or process, in which goals cannot be attained or are attained too slowly. We model how pathologies arise in the execution of simulated transportation plans, and develop visualizations – graphical, model-based displays of pathologies – to detect and correct pathologies during execution of these plans. We begin with the simulated execution of a transportation plan, and rely on pathology demons to detect and predict pathologies as early as possible. Detecting a pathology involves successfully matching a model of a pathology to a history of some of the events in a plan; a graphic visualization of this history is produced as a side-effect of detection. The purpose of IPS is to steer a plan away from a pathology – avoiding the pathology if it has not yet materialized, and disabling it quickly if it has. The model of the pathology is consulted by the IPS system to suggest alternative steering actions. The visualization of the pathology provides an interface between the user and the model of the pathology, allowing the user to explore in a "what if" mode how steering actions – as suggested by the system or of his own design – are predicted to affect the course of the pathology.

Much AI research has been focused on the generation of plans and schedules, but relatively little has examined their execution, particularly for large-scale real-world problems such as transportation planning. The key new idea in this research is to focus on predicting pathological situations that can throw a transportation plan off schedule and helping the user steer the plan around them. The result is a mixed-initiative execution monitoring system that improves the performance of transportation plans.

Our IPS system comprises:

- a pathology demon that monitors the execution environment to detect and predict pathological states,
- a plan steering agent that evaluates the demon's predictions and formulates plan modifications to avoid predicted pathologies, and

- a human user who monitors the environment, the demon, and the agent.

As a first step toward plan steering, we built a plan steering agent for the related task of schedule maintenance in TRANSSIM, our simulation testbed of the the IFD2 transportation planning domain (see Figure 1). We experimentally assessed the performance of the agent at its two primary tasks: predicting schedule pathologies and formulating schedule modifications to avoid those pathologies. In those experiments [1] the agent was completely responsible for managing the schedule; no human intervention was allowed [8] [2]. Next, we evaluated the performance of humans at that same task both with and without the aid of the agent. We showed that the agent can help human planners - indeed, working in concert, humans and an agent perform better than either does alone [9]. The results of these experiments are detailed in Section 2.

TransSim as a Research Tool. TRANSSIM was originally based on the ARPI's Prototype Feasibility Estimator (PFE), but has been rewritten to provide a full simulator, support for the execution of a global schedule, and the tools necessary to support interactive plan steering. A TRANSSIM scenario allows the user to configure an arbitrary shipping network by specifying ports, a set of cargo inputs, and a list of available ships. Input to the scenario is a list of Simple Movement Requirements (SMR), which are based on the TPFDD's of IFD2. SMR's specify a port of embarkation, various intermediate ports, and a port of debarkation. Cargo appears at ports of embarkation at intervals specified by the scenario. An individual piece of cargo travels through the network along the route specified by its SMR. Each port has an associated Port Manager that schedules the unloading of incoming cargo, matches cargo to available ships, and schedules the loading of outgoing cargo. If no ships are available, the Port Manager requests ships with available capacity from other ports, following a simple set of heuristics for making its requests. While TRANSSIM is capable of representing most of the objects and constraints in the IFD2 shipping domain provided by ISX (e.g., ship types, cargo types, port characteristics), we abstract some of these domain features in TRANSSIM for the sake of simulation speed. By so doing we are able to run controlled experiments in large scale scenarios to test the scalability of IPS.

While we developed the TRANSSIM testbed as a vehicle to further our research in IPS, other ARPI participants have expressed interest in using TRANSSIM for purposes that include the development of integrated plan execution and plan repair components and the incorporation of learning methods to improve plan generation, to be achieved by simulating generated plans and using the results to refine operator selection (see Technology Transfer, Section1.6).

How IPS fits into the ARPI Toolbox. Interactive Plan Steering has the potential to offer direct, computer-assisted control over the dynamic unfolding of transportation plans in the USTRANSCOM domain. Other CPE components focus on the development of high level courses of action, which are passed to the CPE's logistical scheduling components to produce full transportation plans. IPS offers a mixed-initiative approach to managing a transportation plan once it is set in motion. While the connections have not been made,

[1] Instrumented using CLIP and analyzed using CLASP (see Section 4).

[2] References for each report section are found at the end of that section.

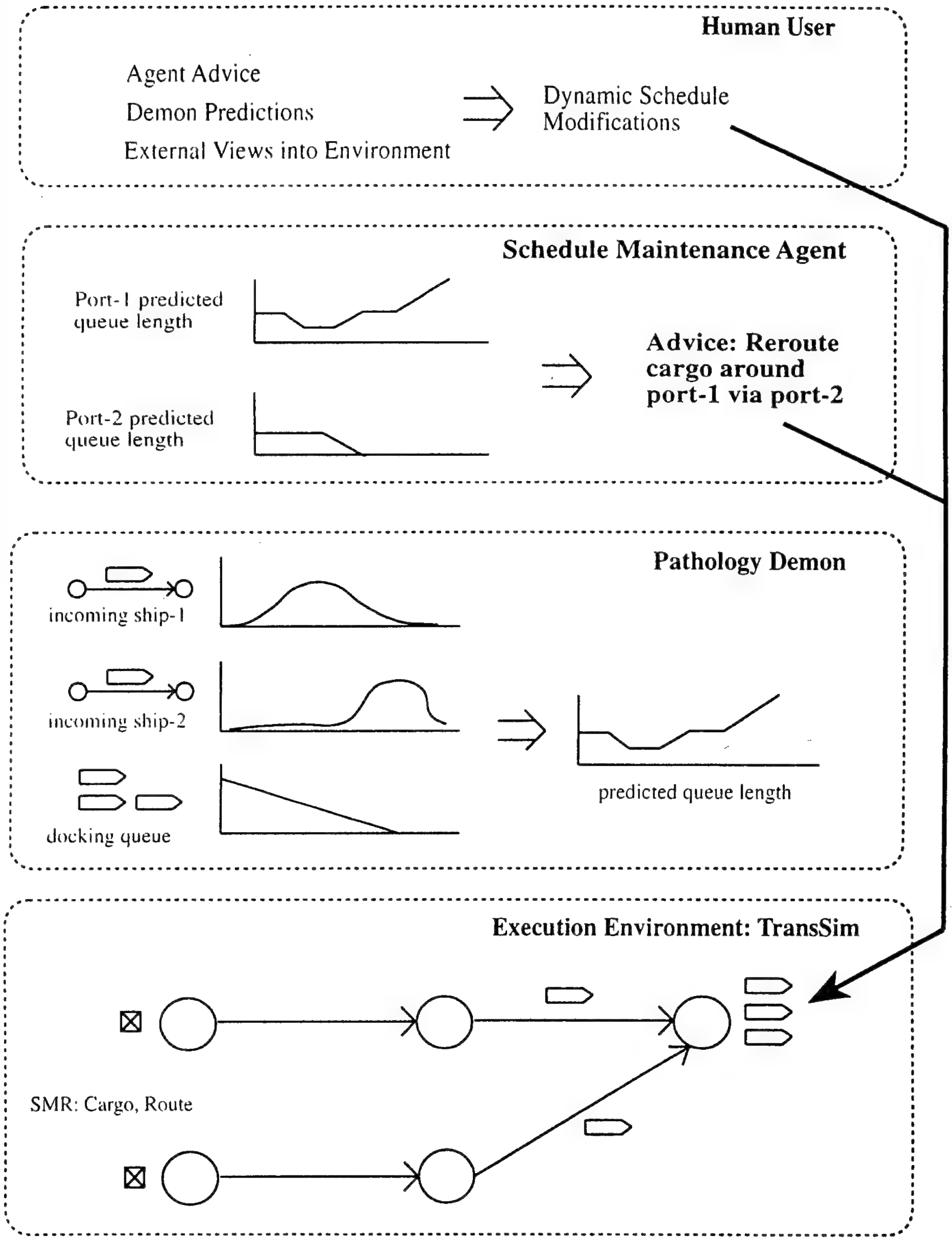

Human User
Agent Advice
Demon Predictions
External Views into Environment
Dynamic Schedule Modifications
Schedule Maintenance Agent
Port-1 predicted queue length
Port-2 predicted queue length
Advice: Reroute cargo around port-1 via port-2
Pathology Demon
incoming ship-1
incoming ship-2
docking queue
predicted queue length
Execution Environment: TransSim
SMR: Cargo, Route

Figure 1: IPS Architecture for Schedule Maintenance in TRANSSIM. At the lowest level is the shipping simulation. Watching the simulation is a Pathology Detection Demon (2nd level up), which models activity in each port to predict traffic bottlenecks. When a pathology is predicted, the Plan Steering Agent assesses the prediction (3rd level). A visualization and steering advice are presented by the PSA to the User (top level), who may accept, modify or ignore the advice.

our system has been *designed* for integration with the logistics scheduling components of the CPE: TRANSSIM can execute a global shipping schedule and provide feedback to the scheduler when the Plan Steering Agent and user, acting in concert, adjust the schedule in response to unfolding events. Indeed, this integration would create a feedback loop in which the plan steering system proposes coarse-grained schedule modifications to the scheduler (such as "divert three ships from Port A to Port B"), which are passed back to the scheduler for fine-grained schedule modification. The modified schedule is then passed back to the plan steering system for evaluation (through simulation). If judged beneficial, the modification is accepted; otherwise, a new modification is proposed and the cycle is repeated until an acceptable steering action is found.

TRANSSIM, as it is, provides a powerful feasibility estimation tool. With the addition of our plan steering architecture we create the capability of intervening in the simulation when the plan goes awry and exploring alternative actions. This enhances TRANSSIM's usefulness as a feasibility estimator, and turns it into an *interactive plan development* tool. Transportation plans generated by other CPE components can be executed under various conditions to test their robustness. Successful modifications developed by the steering system can be incorporated into the plans as they evolve.

For a full description of our IPS work see Section 2.

MSDD: A New Algorithm for Pathology Prediction. We have developed a new technique for pathology detection in TRANSSIM called *Multi-stream Dependency Detection*, or MSDD [10]. This technique is an extension of Adele Howe's thesis work, which identified pathological behaviors in PHOENIX by uncovering significant dependencies between failure events in program execution traces [6, 7]. Whereas Howe used a single execution trace, or stream, we are now using similar statistical techniques to find dependencies in multiple concurrent streams. This is useful in identifying patterns that predict pathologies. For example, a large queue length for several days at two nearby ports in the TRANSSIM network should be a predictor of large queue length at one or more "downstream" ports in the near future. (Each port produces a time-stamped execution stream, with one entry for Queue Length at each time step.) Such a pattern, or pathology signature, would be identified from execution traces over many trials. Once identified, it can be detected by a demon while the system is running.

Identifying such patterns can help us in two ways: First, they will reveal new kinds of pathologies, particularly as the networks scale up to a point where there is too much activity for the knowledge engineer to clearly identify pathology signatures. Second, they will help us design better pathology detection demons, which will be tailored to specific signatures found in the execution traces.

In Section 3 we describe the development and significance of MSDD. We introduce the algorithm, which performs a general-to-specific best-first search over the exponentially sized space of possible dependencies between multi-tokens. The search heuristic employed by MSDD strikes a tunable balance between the expected number of hits and false positives for the dependencies discovered when they are applied as predictive rules to previously unseen data from the same source. We present results from an empirical evaluation of MSDD's performance over a wide range of artificially generated data. In addition, we applied MSDD to two tasks: pathology prediction in TRANSSIM, and classification tasks found in the UC Irvine problems that are used to benchmark Machine Learning algorithms. The results that we obtained, which are detailed in Section 3, are very encouraging.

We are currently working on an incremental version of MSDD that can identify dependencies by processing data as it is generated, and that adapts to changing probability distributions. Also, we are working to remove the need for a fixed-size multi-token and a fixed time interval between multi-tokens. Both of these improvements would make MSDD more useful for the task of pathology detection. This work will continue under our contract for the development of Experimental Methods for Evaluating Planning Systems, and its results will be integrated into the CPE as part of the Dependency Detection Experiment Module.

For more on MSDD see Section 3.

1.2.2 Empirical Tools for Evaluating Planners (CPE Support)

When events occur in the world, people turn to statistics to analyze and describe them. This is especially true of the behavioral sciences, psychology and sociology. Through statistics we attempt to separate those events which are simply due to chance from those which are a result of, or are related to, other events in our world. Through careful analysis it is possible to understand a great deal about the state of the world. In AI, we have a new aspect of behavioral science, the study of the behavior of intelligent agents and computer programs in complex environments, and so we turn to statistical analysis to describe and analyze their behavior and the factors that predict their behavior. Two examples of such AI programs are PHOENIX, a simulator of agents that fight forest fires in Yellowstone National Park, and TRANSSIM (see above). We expect that these simulated worlds will lend themselves to the same forms of analysis we use to study the natural world. Our goal is to smoothly connect the simulators and other computer programs with statistical analysis tools, written in Common Lisp and running in the same Lisp world, so that high quality statistical analysis will be conveniently available to help us understand and test the behavior of our programs.

Why use statistics to understand our programs? Why not just look at the code? As our programs become increasingly large and complex, studying the code becomes as effective in understanding overall behavior as studying biochemistry is in understanding animal behavior. The interaction of components and the response of the program to different circumstances of its execution make global behavior hard to predict. Furthermore, programs will act differently in different executions because of differences in their environments, and statistics can help us summarize, visualize, and analyze the overall behavior, abstracting away the details of particular executions.

To help AI researchers analyze the behavior of their programs, and in particular to support incremental development and empirical evaluation of component technologies for the CPE, we ported and extended our CLIP/CLASP empirical analysis environment to the CPE. CLIP/CLASP was first delivered in October 1993, when we held a very successful workshop at Rome Laboratory to introduce it to ARPI participants. Access to CLIP/CLASP was provided at the top level of the CPE user interface to encourage its use. Since then, several enhanced versions have been delivered. Full documentation, a tutorial and ongoing cross-platform support are provided by e-mail, ftp and WWW.

CLIP/CLASP forms the nucleus of a larger experimental framework we are developing under a separate contract (Experimental Methods for Evaluating Planning Systems). A series of Experiment Modules for evaluating planner behavior are being developed, using CLIP/CLASP as a substrate and user interface. CLIP/CLASP is also used throughout a textbook we have written called *Empirical Methods for Artificial Intelligence* that will be published by MIT Press in 1995. This text was inspired by an NSF/ARPA-sponsored workshop on AI Evaluation Methodology held here in 1990. The text uses many examples of empirical analysis from the AI literature to illustrate the range of available techniques and their appropriate uses. CLIP/CLASP is used in these examples whenever statistical analyses are shown. A version of CLIP/CLASP for the Macintosh will be distributed with the text so that students can recreate the analyses they see and do their homework with the same package.

CLIP/CLASP is an online laboratory environment for the analysis of AI planning systems [1]. Using CLIP/CLASP one can:

- collect and filter data while a planner runs
- define and automatically run experiments to evaluate a planner's behavior in different conditions
- transform, filter and partition data in powerful ways
- examine data graphically and statistically
- run parametric and computer-intensive tests.

CLIP/CLASP is composed of two integrated tools: the Common Lisp Instrumentation Package (CLIP) for data collection and experiment design [11] and the Common Lisp Analytical Statistics Package (CLASP) for data manipulation and statistical analysis [2]. CLIP/CLASP was developed in our laboratory to support the empirical analysis of AI planners in the ARPI, and is now being used in a number of AI laboratories.

Using CLIP the experimenter defines and manages "alligator clips" to collect data while the planner runs. CLIP writes this data to files in a format specified by the experimenter, who may choose from among a variety of commonly used data formats or use a format customized for CLASP analysis. Once defined, clips may be saved for reuse in other experiments.

CLIP also includes facilities to support experiment design, such as definition of experimental factors, scenario scripting, and automatic factorial combination of independent variables. CLIP automatically collects summary data at the end of each experiment trial, and can also collect and filter time-series and event-based data for periodic and/or non-periodic events specified by the experimenter. CLIP allows partially completed experiments to be restarted from the point of interruption.

CLASP provides an interactive environment for data manipulation and statistical analysis that is fully integrated with Common Lisp. CLASP has all the descriptive and hypothesis-testing statistics one expects of a moderately powerful statistics package, plus it includes many features that facilitate exploratory data analysis. CLASP also provides powerful facilities for data manipulation, assessing confidence intervals and testing hypotheses. It has a clearly-defined programmer interface and can be extended by the user with the addition of new features.

More specific information about CLIP/CLASP can be found in Software Prototypes (Section 1.7) and in Tools for Experiments in Planning (Section 4). Full documentation may be found in the CLASP and CLIP User Manuals [2, 11].

1.2.3 Investigating Monitoring Strategies for Embedded Agents

During the course of this contract we also explored monitoring strategies for real-time plan execution under an AASERT award. This research, which followed two complementary directions, grew out of a realization that current monitoring strategies employed in autonomous systems are simplistic and often not cost effective. In one line of research we identified a monitoring strategy that we showed to be superior for an important class of problems to the one commonly employed in planning and robotic systems – periodic monitoring [4, 3]. In related work we developed an efficient, anytime planner that controls the rate of sensing during plan execution [5]. Both of these results are significant for the design of embedded agents for real-world task environments.

The Interval Reduction Monitoring Strategy. Monitoring is the process by which agents assess their environments. Most AI applications rely on periodic monitoring, but for a large class of problems this is inefficient. For this class of problems, in which a planner must monitor a plan step and determine the appropriate time to end it, the *interval reduction* monitoring strategy is better. The interval reduction strategy reduces the time interval between monitoring events as the plan step nears completion. This strategy also appears in humans and artificial agents when they are given the same set of monitoring problems. We implemented two genetic algorithms to evolve monitoring strategies and a dynamic programming algorithm to find an optimum strategy. We also developed a simple mathematical model of monitoring. We tested all these strategies in simulations, and we tested human strategies in a "video game." Results of these tests show that interval reduction always emerges [4]. Environmental factors such as error and monitoring costs had the same qualitative effects on the strategies, irrespective of their genesis. Interval reduction appears to be a general monitoring strategy.

We have recently extended this work, demonstrating mathematically that interval reduction outperforms periodic monitoring for this class of problems [3]. We also show how features of the environment may influence the choice of the optimal strategy. This paper concludes with some thoughts about a monitoring strategy taxonomy, and what its defining features might be. This work is summarized in Monitoring in Embedded Agents (Section 5).

Cost-Effective Sensing During Plan Execution. Between sensing the world after every action (as in a reactive plan) and not sensing at all (as in an open-loop plan), lies a continuum of strategies for sensing during plan execution. If sensing incurs a cost (in time or resources), the most cost-effective strategy is likely to fall somewhere between these two extremes. Yet most work on plan execution assumes one or the other. We have developed an efficient, anytime planner is described that controls the rate of sensing during plan execution [5]. The sensing interval is determined by the state during plan execution, as well as by the cost of sensing, so that an agent can sense more often when necessary. The planner is based on a generalization of stochastic dynamic programming. This work is summarized in Cost Effective Sensing During Plan Execution (Section 6).

1.3 Publications, Reports and Articles

1.3.1 Refereed Papers Published

Tim Oates and Paul R. Cohen. 1994. Toward a Plan Steering Agent: Experiments with Schedule Maintenance. *Proceedings, Second International Conference on Artificial Intelligence Planning Systems.* AAAI Press, pp. 134-139.

Adele E. Howe and Paul R. Cohen. Understanding Planner Behavior. To appear in *AI Journal*, Winter 1995.

Adele E. Howe. 1992. Analyzing Failure Recovery to Improve Planner Design. In *Proceedings of the Tenth National Conference on Artificial Intelligence.* AAAI Press/MIT Press, pp. 387-392.

Paul R. Cohen, Robert St. Amant, and David M. Hart. 1992. Early Warnings of Plan Failure, False Positives and Envelopes: Experiments and a Model. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society.* Lawrence Earlbaum Associates, Inc., pp. 773-778.

David M. Hart and Paul R. Cohen. 1992. Predicting and Explaining Success and Task Duration in the Phoenix Planner. In *Artificial Intelligence Planning Systems: Proceedings of the First International Conference (AIPS92).* Morgan Kaufmann Publishers, Inc., pp. 106-115.

Paul R. Cohen, Adam Carlson, Lisa Ballesteros and Robert St. Amant. 1993. Automating Path Analysis for Building Causal Models from Data. In *Proceedings of the Tenth International Conference on Machine Learning.* Morgan Kaufmann Publishers, Inc., pp. 57-64.

Scott D. Anderson, Adam Carlson, David L. Westbrook, David M. Hart and Paul R. Cohen. 1994. Tools for Experiments in Planning. In *Proceedings of the Sixth IEEE International Conference on Tools with AI.* IEEE Computer Society Press, pp. 615-623.

Steve Hanks, Martha E. Pollack and Paul R. Cohen. 1993. Benchmarks, Testbeds, Controlled Experimentation, and the Design of Agent Architectures. In *AI Magazine*, Vol. 14, No. 4, pp. 17-42.

Paul R. Cohen, Marc S. Atkin and Eric A. Hansen. 1994. The Interval Reduction Strategy for Monitoring Cupcake Problems. In *Proceedings, From Animals to Animats, the Third International Conference on Simulation of Adaptive Behavior.* MIT Press, pp. 82-90.

Eric A. Hansen. 1994. Cost-Effective Sensing During Plan Execution. In *Proceedings of the Twelfth National Conference on Artificial Intelligence.* AAAI Press/MIT Press, pp. 1029-1035.

Marc S. Atkin and Paul R. Cohen. 1994. Learning Monitoring Strategies: A Difficult Genetic Programming Application. In *Proceedings, IEEE International Conference on Evolutionary Computation.* Piscataway NJ: IEEE, pp. 328-332.

1.3.2 Refereed Papers Submitted

Tim Oates, Dawn E. Gregory and Paul R. Cohen. 1995. Detecting Complex Dependencies in Categorical Data. Submitted to *Fourteenth International Joint Conference on Artificial Intelligence* (IJCAI-95).

Mark S. Atkin and Paul R. Cohen. 1995. Monitoring in Embedded Agents. Submitted to *Fourteenth International Joint Conference on Artificial Intelligence* (IJCAI-95).

1.3.3 Invited Papers Published

Tim Oates and Paul R. Cohen. 1994. Mixed-Initiative Schedule Maintenance: A First Step Toward Plan Steering. In *Workshop Proceedings, ARPA/Rome Laboratory Knowledge-Based Planning and Scheduling Initiative*, M.H. Burstein, Ed. Morgan Kaufmann Publishers, Inc., pp. 133-144.

Scott D. Anderson, Adam Carlson, David L. Westbrook, David M. Hart and Paul R. Cohen. 1994. Tools for Experiments in Planning. In *Workshop Proceedings, ARPA/Rome Laboratory Knowledge-Based Planning and Scheduling Initiative*, M.H. Burstein, Ed. Morgan Kaufmann Publishers, Inc., pp. 423-432.

Paul R. Cohen and Marc S. Atkin. 1994. The Interval Reduction Strategy for Monitoring Cupcake Problems. In *Workshop Proceedings, ARPA/Rome Laboratory Knowledge-Based Planning and Scheduling Initiative*, M.H. Burstein, Ed. Morgan Kaufmann Publishers, Inc., pp. 15-26.

1.3.4 Refereed Workshop Abstracts and Symposia Papers

Tim Oates, Dawn E. Gregory and Paul R. Cohen. 1995. Detecting Complex Dependencies in Categorical Data. In *Preliminary Papers of the Fifth International Workshop on Artificial Intelligence and Statistics*, pp. 417-423.

Scott D. Anderson, Adam Carlson, David L. Westbrook, David M. Hart and Paul R. Cohen. 1995. Tools for Empirically Analyzing AI Programs. In *Preliminary Papers of the Fifth International Workshop on Artificial Intelligence and Statistics*, pp. 35-41.

Marc S. Atkin and Paul R. Cohen. 1993. Genetic Programming to Learn an Agent's Monitoring Strategy. In *Working Notes of the Learning Action Models Workshop*, Eleventh National Conference on Artificial Intelligence, Washington DC, pp. 36-41.

Eric A. Hansen and Paul R. Cohen. 1993. Learning Monitoring Strategies to Compensate for Model Uncertainty. In *Working Notes of the Learning Action Models Workshop*, Eleventh National Conference on Artificial Intelligence, Washington DC, pp. 33-35.

Paul R. Cohen and David M. Hart. 1993. Path Analysis Models of an Autonomous Agent in a Complex Environment. In *Preliminary Papers of the Fourth International Workshop on AI and Statistics*, pp. 185-189.

Paul R. Cohen, Lisa Ballesteros, Adam Carlson, and Robert St. Amant. 1993. Automating Path Analysis for Building Causal Models from Data: First Results and Open Problems. In *Working Notes of the Knowledge Discovery in Databases Workshop*, Eleventh National Conference on Artificial Intelligence, Washington DC, pp. 153-161.

Paul E. Silvey, Cynthia L. Loiselle and Paul R. Cohen. 1992. Intelligent Data Analysis. In *Working Notes of the Intelligent Scientific Computing Workshop*, AAAI-92 Fall Symposium, Boston, MA.

1.3.5 Books or Parts Thereof Published

Paul R. Cohen. *Empirical Methods for Artificial Intelligence.* The MIT Press, in press, 1995.

Paul R. Cohen, David M. Hart, Robert St. Amant, Lisa A. Ballesteros, and Adam Carlson. 1994. Path Analysis Models of an Autonomous Agent in a Complex Environment. In *Selecting Models from Data: Artificial Intelligence and Statistics IV*, Peter Cheeseman and R.W. Oldford (Eds.). Springer-Verlag, pp. 243-251.

Adele E. Howe and Paul R. Cohen. 1994. Detecting and Explaining Dependencies in Execution Traces. In *Selecting Models from Data: Artificial Intelligence and Statistics IV*, Peter Cheeseman and R.W. Oldford (Eds.). Springer-Verlag, pp. 71-77.

1.3.6 Ph.D. Dissertations

Adele E. Howe. February 1993. Accepting the Inevitable: The Role of Failure Recovery in the Design of Planners.

1.3.7 Unrefereed Reports and Articles

Tim Oates and Paul R. Cohen. 1994. Evaluation of a Mixed-Initiative Approach to Schedule Maintenance. Technical Report 95-08. Department of Computer Science, University of Massachusetts/Amherst.

Tim Oates and Paul R. Cohen. 1994. Humans Plus Agents Maintain Schedules Better than Either Alone. Technical Report 94-03. Department of Computer Science, University of Massachusetts/Amherst.

Tim Oates. 1994. MSDD as a Tool for Classification. EKSL Memo #94-29, Department of Computer Science, University of Massachusetts/Amherst.

David L. Westbrook, Scott D. Anderson, David M. Hart and Paul R. Cohen. 1994. Common Lisp Instrumentation Package: User Manual. Technical Report 94-26. Department of Computer Science, University of Massachusetts/Amherst.

Paul R. Cohen. 1993. Statistics Short Course: A Tutorial on Exploratory Data Analysis and Hypothesis Testing. Presented at CLIP/CLASP Workshop, Rome Laboratory, Griffiss AFB, NY.

Paul R. Cohen, David Westbrook and Adam Carlson. 1993. CLIP/CLASP Workshop. Conducted for Planning Initiative participants at Rome Laboratory, Griffiss AFB, NY.

Scott D. Anderson, Adam Carlson, David L. Westbrook, David M. Hart, and Paul R. Cohen. 1993. Common Lisp Analytical Statistics Package: User Manual. Technical Report 93-55. Department of Computer Science, University of Massachusetts/Amherst.

Eric A. Hansen and Paul R. Cohen. 1992. Learning a Decision Rule for Monitoring Tasks with Deadlines. Technical Report 92-80. Department of Computer Science, University of Massachusetts/Amherst.

1.4 Conferences, Workshops and Presentations

1.4.1 Invited Presentations

October 1991: Paul Cohen served as a panel member in the opening session of the *Workshop on Research in Experimental Computer Science* chaired by Barbara Liskov in Palo Alto, CA;

December 1991: Served as a panel member for "The Future of Expert Systems" chaired by Dr. Y.T. Chen of NSF at the World Congress on Expert Systems in Orlando, FL.

He also gave the following invited talks:

- March 1992: "Three Examples of Statistical Modeling of an AI Program," at the University of Texas, Austin.
- April 1992: "Methods for Agentology: General Concerns, Specific Examples," at Virginia Polytechnic Institute.
- April 1992: "Methods for Studying Agents: General Issues and Specific Examples," at the U.S. Navy Center for Applied Research in Artificial Intelligence, Naval Research Laboratory.
- May 1992: "Methods for Agentology: General Concerns, Specific Examples," at the University of West Virginia.
- December 1992: "Timing is Everything," at the United Technologies Day Artificial Intelligence Workshop, University of Massachusetts/Amherst.
- March 1993: "A Research Proposal for an Online laboratory: Statistical Techniques for Modeling Computer Programs," at the Rutgers University Graduate School of Management.
- June 1994: "Using Statistics in AI," at the *Ninth Annual Conference: Making Statistics More Effective in Schools of Business*, Rutgers University Graduate School of Management.

June 1992: Paul Cohen, David Hart and Adele Howe attended the First International Conference on Artificial Intelligence Planning Systems at the University of Maryland, College Park, where Hart presented a paper entitled "Predicting and Explaining Success and Task Duration in the Phoenix Planner;" Cohen served on a panel entitled "The Empirical Evaluation of Planning Sytems: Promises and Pitfalls."

July 1992: Paul Cohen and Adele Howe attended the Tenth National Conference on Artificial Intelligence in San Jose, CA where Howe presented her paper, "Analyzing Failure Recovery to Improve Planner Design;" Cohen served on a panel entitled "Planning Under Uncertainty," at the *SIGMAN Workshop on Knowledge-Based Production Planning, Scheduling and Control.*

October 1992: Adam Carlson, Paul Cohen, Cynthia Loiselle and Paul Silvey attended the AAAI Fall Symposium on *Intelligent Scientific Computing* in Boston, MA; Cohen presented a paper entitled "Inteligent Data Analysis."

January 1993: Paul Cohen and David Hart attended the *Fourth International Conference on Artificial Intelligence and Statistics* in Fort Lauderdale, FL where they presented their work on *path analysis* (Hart and Cohen 1992, Cohen and Hart 1993). Adele Howe (subcontractor on our experimental methods contract) also attended and presented her work on analyzing planner behavior using *dependency detection.*

February 1993: Paul Cohen, David Hart and David Westbrook attended the ARPA/Rome Laboratory Annual Planning Initiative Workshop in San Antonio, TX, where Cohen gave an invited talk, "NEO Phoenix: A Testbed for the Planning Initiative." Hart and Westbrook presented a 10-minute video of the CLIP/CLASP system and distributed a range of documentation materials.

April 1993: Paul Cohen served as an invited member of the Young Investigator Award Review Panel, CISE Directorate of the National Science Foundation.

June 1993: Paul Cohen presented the paper, "Automating Path Analysis for Building Causal Models from Data" at the *Tenth International Conference on Machine Learning* at the University of Massachusetts, Amherst, MA.

July 1993: EKSL was represented at the *Eleventh National Conference on Artificial Intelligence* in Washington, DC by Paul Cohen, David Hart, David Westbrook, Adam Carlson, and five graduate students.

- Paul Cohen organized and moderated a panel which included Lynette Hirschman (Mitre Corp.), Drew McDermott (Yale Univ.), Bruce Porter (UTexas/Austin) and Charles Weems (UMass/Amherst) entitled "The Pros and Cons of Evaluation."
- "Genetic Programming to Learn an Agent's Monitoring Strategy" and "Learning Monitoring Strategies to Compensate for Model Uncertainty" were presented at the *Learning Action Models Workshop.*
- "Automating Path Analysis for Building Causal Models from Data: First Results and Open Problems" was presented at the *Knowledge Discovery in Databases Workshop.*

June 1994: Paul Cohen and Tim Oates attended the Second International Conference on Artificial Intelligence Planning Systems in Chicago, IL, where Oates presented the paper, "Toward a Plan Steering Agent: Experiments with Schedule Maintenance."

June 1994: Marc Atkin attended *ICEC '94, World Congress on Computational Intelligence* in Orlando, FL. He presented the paper, "Learning Monitoring Strategies: A Difficult Genetic Programming Application."

August 1994: Eric Hansen attended the *Twelfth National Conference on Artificial Intelligence* and presented the paper, "Cost-Effective Sensing During Plan Execution."

March and April 1992: Adele Howe gave an invited talk entitled "Accepting the Inevitable: The Role of Failure Recovery in the Design of Planners," at the University of Maryland, Baltimore County; at Purdue University, Oregon State University, Carnegie Mellon University, Clarkson University, and Colorado State University.

1.4.2 Contributed Presentations

November 1991: Paul Cohen participated in the Planning Initiative's workshop in Chicago as a speaker and as an Issues Group Co-Chair.

March 1992: Paul Cohen and Adele Howe attended the AAAI Spring Symposium on *Computational Considerations in Incremental Plan Modification and Reuse* in Palo Alto, CA, where Howe gave a contributed presentation entitled "Failure Recovery Analysis as a Tool for Plan Debugging."

August 1992: EKSL hosted a site visit by Nort Fowler, Lou Hoebel and Don Roberts of Rome Laboratory. Demonstrations included shipping simulation, experimental interface (including alligator clips and data collection methods), simple visualizations, statistical analysis tools, demons detecting pathologies in a sample eight-port scenario, and the introduction of a constraint-based scheduler developed by another UMass research group.

August 1992: Paul Cohen participated as a member of a Study Group for the Information Science and Technology (ISAT) Advisory Group on Simulation, Institute for Defense Analysis, held at Woods Hole, MA.

October 1992: David Hart and David Westbrook attended the Technology Integration Experiments at Rome Laboratory, Griffiss AFB, NY.

October 1993: EKSL organized and presented the "CLIP/CLASP Workshop" hosted by Rome Laboratory for Planning Initiative participants, Griffiss AFB, NY, which included:

- Two tutorials conducted by Paul Cohen: "Statistics Short Course: A Tutorial on Exploratory Data Analysis and Hypothesis Testing," and an on-line "Tutorial in the Use of CLIP/CLASP."
- A presentation by David Westbrook on the use of CLIP.
- A technical introduction to CLASP presented by Adam Carlson.

December 1993: Paul Cohen attended the quarterly Planning Initiative meeting at Yale University, where he helped plan and lead part of the meeting devoted to evaluation of the PI's research results.

February 1994: Paul Cohen, David Hart, David Westbrook (UMass), and Adele Howe (Colorado St. Univ.) attended the ARPA/Rome Laboratory Knowledge-Based Planning and Scheduling Initiative Workshop in Tucson, AZ and participated in the following:

- A panel led by Paul Cohen describing the "Handbook of Evaluation for the ARPA/Rome Lab Planning Initiative."
- A demo led by Hart and Westbrook of our interactive plan steering system (Oates et al. 1994).
- A paper presented by Hart on plan steering in the transportation planning domain (Oates and Cohen 1994).

- A poster presentation by Hart on our AASERT work on strategies for monitoring (Cohen et al. 1994).
- A "CLIP/CLASP Workshop" led by Cohen and Westbrook.
- A paper presented by Howe, "Statistical Methods for Characterizing Causal Influences on Planner Behavior Over Time."

May 1994: Paul Cohen hosted a visit by Prof. Steve Hanks from the University of Washington who gave a talk to the department on enhancing AI planning representations to support probabilistic reasoning and temporal projection. EKSL demonstrated our plan steering system for Prof. Hanks.

1.4.3 Tutorials

July 1992: Paul Cohen and Bruce Porter (University of Texas, Austin) presented a tutorial entitled "Experimental Methods for Evaluating AI Systems," at the *Tenth National Conference on Artificial Intelligence,* San Jose, CA.

July 1993: Paul Cohen and Bruce Porter repeated their tutorial at the *Eleventh National Conference on Artificial Intelligence,* Washington, DC.

October 1993: Paul Cohen presented two tutorials at the "CLIP/CLASP Workshop" held at Rome Laboratory, Griffiss AFB:

- "Statistics Short Course: A Tutorial on Exploratory Data Analysis and Hypothesis Testing."
- "Tutorial in the Use of CLIP/CLASP."

1.5 Awards, Promotions, Honors

Paul Cohen was elected Fellow of the American Association for Artificial Intelligence in 1993, and served as Councillor of that organization from 1991 to 1994. In that capacity he was appointed to the following Committees:

- Chair, Tutorial Committee, 1992-1993.
- Assistant to the Co-Chair, Program Committee, 1992-1993.
- Co-Chair, Symposium Committee, 1992-1993.
- Co-Chair, Tutorial and Symposium Committees, 1991-1992.

Cohen also served as a member of the Program Committee for the Fifth International Workshop on Artificial Intelligence and Statistics, held in Fort Lauderdale, FL in January, 1995.

Adele Howe was appointed Assistant Professor of Computer Science at Colorado State University, Fort Collins, CO in September, 1993. Howe received her Ph.D. in 1993, and was nominated for the ACM Distinguished Dissertation Award by the Computer Science Department, UMass/Amherst.

Marc Atkin, Eric Hansen, Sameen Fatima, Robert St. Amant and Paul Silvey, graduate students in the Experimental Knowledge Systems Laboratory, all achieved their M.S. degrees in 1993.

1.6 Technology Transfer

CLIP/CLASP has been installed in the ARPI's CPE and plans called for its installation in Rome Laboratory's Advanced AI Technology Testbed (AAITT). It is also being used in numerous other research labs. Workshops were held in 1993 (Rome Laboratory) and 1994 (Tucson ARPI meeting) to introduce CLIP/CLASP to the ARPI community. As mentioned above, a Macintosh version of CLIP/CLASP will be distributed with our forthcoming textbook for use by students and instructors.

TRANSSIM and our IPS system have been provided to Edinburgh's AI Applications Institute and to the Stanford Research Institute where it is being considered for use as a simulator for O-PLAN2 and for SIPE/SOCAP.

1.7 Software Prototypes

CLIP/CLASP is an online laboratory environment for the analysis of AI planning systems. It comprises two integrated tools: the Common Lisp Instrumentation Package (CLIP) for data collection and experiment design, and the Common Lisp Analytical Statistics Package (CLASP) for data manipulation and statistical analysis.

CLIP/CLASP is written in Common Lisp and CLOS. A portable version runs on several Unix workstations. Its interface, which uses the Common Lisp Interface Manager (CLIM), combines the desktop-style accessibility of all data objects with full integration into the underlying lisp environment. Commands are menu selectable or can be entered by hand. Command arguments, such as datasets, variables, previous results, and even graphic objects, are all mouse sensitive. All session activity appears in a notebook that can be saved for future use or record-keeping. A Macintosh (MCL) version of CLIP/CLASP includes much of the functionality of the Unix version while providing a native Macintosh interface.

CLIP/CLASP is distributed with a User Manual and with a tutorial for CLASP. The Unix version of CLIP/CLASP is part of the CPE and is also available for non-commercial use through anonymous ftp at ftp.cs.umass.edu. CLIP can be found under the directory pub/eksl/clip, CLASP under pub/eksl/clasp, and a tutorial on CLASP under pub/eksl/clasp-tutorial. Ongoing, cross-platform support are provided by e-mail, ftp and World Wide Web.

CLIP/CLASP has been fully integrated into the ARPI Common Prototyping Environment (see Section 1.2.2).

References

[1] Anderson, S.D., A. Carlson, D.L. Westbrook, D.M. Hart and P.R. Cohen. Tools for Experiments in Planning. In *Proceedings of the Sixth International IEEE Conference on Tools with Artificial Intelligence.* IEEE Computer Society Press, pp. 615-623. 1994.

[2] Anderson, S.D., A. Carlson, D.L. Westbrook, D.M. Hart and P.R. Cohen. Common Lisp Analytical Statistics Package: User Manual. Technical Report 93-55, Computer Science Department, University of Massachusetts/Amherst. 1993.

[3] Atkin, M.S. and Cohen, P.R. Monitoring in Embedded Agents. Submitted to *Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95).* 1995.

[4] Cohen, P.R., M.S. Atkin and E.A. Hansen. The Interval Reduction Strategy for Monitoring Cupcake Problems. In *Workshop Proceedings, ARPA/Rome Laboratory Knowledge-Based Planning and Scheduling Initiative* (M.H. Burstein, Ed.). Morgan Kaufmann, pp. 15-25. 1994.

[5] Hansen, E.A. Cost-Effective Sensing During Plan Execution. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94).* AAAI Press/The MIT Press, pp. 1029-1035. 1994.

[6] Howe, A.E. and P.R. Cohen. Understanding Planner Behavior. To appear in *AI Journal*, Winter 1995.

[7] Howe, A.E. Accepting the Inevitable: The Role of Failure Recovery in the Design of Planners. Ph.D. Thesis. Technical Report 93-40, Computer Science Department, University of Massachusetts/Amherst. 1993.

[8] Oates, T. and P.R. Cohen. Toward a Plan Steering Agent: Experiments with Schedule Maintenance. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pp. 134-139, 1994.

[9] Oates, T. and P.R. Cohen. Mixed-Initiative Schedule Maintenance: A First Step Toward Plan Steering. In *Workshop Proceedings, ARPA/Rome Laboratory Knowledge-Based Planning and Scheduling Initiative*, M.H. Burstein, Ed. Morgan Kaufmann Publishers, Inc., pp. 133-144. 1994.

[10] Oates, T., D.E. Gregory and P.R. Cohen. Detecting Complex Dependencies in Categorical Data. Submitted to *Fourteenth International Joint Conference on Artificial Intelligence* (IJCAI-95). 1995.

[11] Westbrook, D.L., S.D. Anderson, D.M. Hart and P.R. Cohen. Common Lisp Instrumentation Package: User Manual. Technical Report 94-26, Computer Science Department, University of Massachusetts/Amherst. 1994.

2 Mixed-Initiative Schedule Maintenance: A First Step Toward Plan Steering

Tim Oates and Paul R. Cohen

Abstract

When a plan involves hundreds or thousands of events over time it can be difficult or impossible to tell whether those events are unfolding "according to plan" and to assess the impact of dynamic plan modifications. Pathological states may arise in which goals cannot be attained or are attained too slowly. Plan steering is an agent-based approach to this problem. The agent monitors an unfolding plan, detects and predicts pathological situations, and develops dynamic plan modifications that will steer the plan around the problem. We present results for a system that performs the related task of schedule maintenance in the transportation planning domain. We evaluate system performance at pathology prediction and pathology avoidance and show that the agent, using limited domain knowledge and simple heuristics, is able to improve throughput significantly. We describe experiments in which humans perform the same schedule maintenance task both with and without the aid of the agent, and show that the human and the agent working together achieve better results than either working alone.

2.1 Introduction

Plans formulated to run in the real world will often fail due to the complexity and unpredictability of the environment. Existing methods to deal with this problem include real time recovery from plan failures [1] [3] [12] and post-hoc plan repair based on failures observed while executing the plan [9]. Failure recovery mechanisms, such as replanning, can be expensive, and it may not be feasible to repair a plan by letting it repeatedly fail. An alternative strategy is to monitor the execution of the plan, attempting to predict pathological states that make it difficult or impossible to achieve goals [8]. Doing so admits the possibility of effecting plan modifications in real time to avoid pathological states.

Plan steering is a mixed-initiative approach to real time prediction and avoidance of plan failures. A plan steering system comprises a pathology demon that monitors the execution environment to detect and predict pathological states, a plan steering agent that evaluates the demon's predictions and formulates plan modifications to avoid predicted pathologies, and a human user who monitors the environment, the demon, and the agent. The human and the agent work together to steer the plan away from potential problems by intervening before they develop. The benefits of keeping computers in the loop are clear. For large, complex plans, involving hundreds or thousands of events over time, determining whether events are unfolding according to plan and assessing the impact of dynamic plan modifications are impossible for humans.

As a first step toward plan steering, we built an agent for the related task of schedule maintenance in the transportation planning domain. We experimentally assessed the perfor-

In *Workshop Proceedings, ARPA/Rome Laboratory Knowlege-Based Planning and Scheduling Initiative*, 1994 (M.H. Burstein, Ed.). Morgan Kaufmann, pp. 133-144.

mance of the agent at its two primary tasks: predicting schedule pathologies and formulating schedule modifications to avoid those pathologies. In those experiments the agent was completely responsible for managing the schedule; no human intervention was allowed. Next, we evaluated the performance of humans at that same task both with and without the aid of the agent. We show that the agent can help human planners - indeed, working in concert, humans and an agent perform better than either does alone.

2.2 The Schedule Maintenance System

The architecture of a generic schedule maintenance system is shown in Figure 2. A pathology demon monitors the environment as a schedule unfolds, detecting and predicting pathological situations. The schedule maintenance agent monitors the demon's output and formulates schedule modifications that address the problems found by the demon. A human user evaluates the agent's advice and effects schedule changes when appropriate. The architecture for plan steering is identical, only the task changes.

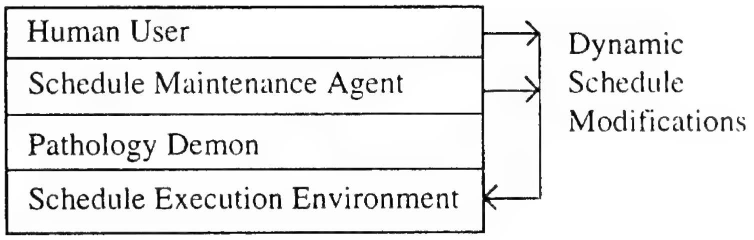


Figure 2: Schedule Maintenance Architecture

The task for our system is management of schedules in a simulated shipping network called TransSim. TransSim is capable of representing most of the objects and constraints in the IFD2 shipping domain provided by ISX, although we abstract some of those domain features for the sake of simulation speed. A TransSim scenario consists of ships, ports, cargo, and simple movement requirements (SMRs) for each piece of cargo. An SMR specifies the route that a piece of cargo is to take through the network and when it is to begin its journey. The SMRs of a scenario constitute its schedule and largely determine the behavior of the simulation. They may be thought of as the schedule generated by a separate scheduling program that must be adhered to as closely as possible and are based on the TPFDD's of IFD2. The scenarios used in experiments throughout this paper enforce constraints on the types of cargo that both ships and ports can accommodate. A typical scenario consists of seven ports, twenty eight ships, and forty SMRs.

If many SMRs reference any one port then it is likely that a *bottleneck* will develop at that port. Ports are limited resources and ships must queue for service when a port is being used to load or unload another ship's cargo. A bottleneck exists at a port when the docking queue at that port is "large" and results in reduced throughput. The goal of the schedule maintenance system is to maximize throughput. It does so by predicting the occurrence of bottlenecks at each port in a scenario and making changes to SMRs where appropriate. The system attempts to minimize the number of changes to preserve as much of the structure imposed by the initial SMRs as possible. These two goals are often at odds with one another so an appropriate balance must be found.

A pathology demon has been implemented to monitor TransSim as cargo is shipped about. It uses simple domain information to predict when and where in the network a bottleneck is likely to arise. The demon looks at the current state of each port and ships that are traveling in channels toward the port, and assigns a probability to each possible state of the port on each day out to a horizon. That is, the demon only uses information local to a given port. Resource-based schedule revision with local information was used successfully in [10]. The schedule maintenance agent combines the demon's predictions for multiple days in the future to determine which ports are likely to become substantial problems. The agent then uses simple heuristics to generate advice that, when implemented, will either alleviate or avoid the predicted bottleneck. This may be contrasted with reactive approaches, such as [7], that respond to unexpected events at the time they occur.

Currently, the only advice the agent offers is based on a simple rerouting heuristic. If a piece of cargo is being loaded onto a ship bound for a potential bottleneck, the agent changes the cargo's SMR so that it travels to the port closest to the original destination that is not problematic. This seems to be a reasonable approach in that rerouted cargo continues to make progress toward its destination. We assume that ports that are close geographically are "equivalent" in some sense.

2.2.1 Performance Assessment of System Components

We now focus on assessing the ability of our system to manage the TransSim domain. First, we will investigate performance of the pathology demon as influenced by environmental factors. Then, we will determine whether the agent's advice is helpful, harmful, or neither. We will also explore how much of the benefit is due to the intelligence built into the agent and how much is due to other factors. Finally, we want to ascertain to what extent environmental factors, such as pathology severity and problem complexity, affect the agent's performance. The general procedure is to run several simulations in each of a variety of experimental conditions, taking several dependent cost measures which are then compared to determine the effect of the condition [5]. Experiments in which humans manage the TransSim domain both with and without the aid of the agent are presented in Section 2.3.

2.2.2 Pathology Demon

The pathology prediction demon models each ship as a probability distribution of arrival times. Combining these distributions with the current state of each port, the demon arrives at a predicted docking queue length for each port. The model is similar to that used in [4] for exploring the effects of resource allocation decisions. Long docking queues indicate the presence of a bottleneck. Several factors affect the accuracy of the demon's predictions.

They are (1) the distance into the future for which predictions are made, (2) the certainty of the demon's knowledge about ship arrivals, and (3) a prediction threshold that controls how aggressive the demon is at making predictions (low values are aggressive, high values are conservative). We expect predictions to be less accurate when made far into the future or when there is a lot of variance in ship arrivals. There should be an optimal prediction threshold, at least for a given level of the other two factors.

The accuracy of the demon was explored by running a fully factorial experiment with ten simulations for each of three levels of the factors (a total of 270 trials). The demon's predictions for each port as well as the actual state of each port were recorded and average squared prediction error was calculated for each trial. For predictions 2, 4, and 6 days into the future, average squared error was 0.137, 0.244, and 0.214; increasing but not monotonically. Though the effect of variance in the demon's knowledge about ship arrivals was not a significant main effect, it did interact significantly with prediction distance. High variance had an increasingly adverse effect on error as prediction distance was increased. Finally, plotting error at several prediction threshold values resulted in a bowl shaped curve with 0.2 being the optimal threshold.

2.2.3 Measures of Cost

Having established some influences on demon accuracy, we turned next to the schedule maintenance agent and its performance. We defined several measures of cost associated with a single run of TransSim. These were used to evaluate the effect of moving from one experimental condition to another.

Bottleneck Predictions The sum over all days of the number of ports marked as potential trouble spots by the demon for a given day.

Cargo Transit The sum over all pieces of cargo of the number of days from when the item first arrived at its port of embarkation until it reached its final destination.

Idle Cargo Cargo is considered idle if it is ready to be loaded onto a ship but none is available or if it is sitting in a ship queued for docking. This measure is the sum over all days of the number of pieces of idle cargo.

Queue Length Each port has a limited number of docks. Ships waiting to dock are placed in the docking queue. This measure is the sum over all days and all ports of the number of ships queued for docking.

Simulated Days The number of simulated days required to ship all cargo to its final destination.

Ship Utility Ships may travel empty when called from one port to another to service cargo. This measure is the sum over all simulated days of the number of ships traveling empty on a given day.

When the agent is not offering advice, we expect a positive correlation between Bottleneck Predictions and many of the other measures such as Queue Length. If the demon is predicting many bottlenecks, and the agent is not doing anything to avoid them, then if the demon is accurate our other cost measures will rise accordingly. The agent was designed to predict and avoid large docking queues so we expect its actions to reduce Queue Length. One extreme way to reduce Queue Length is to let there be only one ship traveling at any time. Both Cargo Transit and Simulated Days provide another view into the agent's performance on a global scale so we may ensure that it is not adopting a similarly pathological strategy.

2.2.4 Agent Advice vs No Advice

Several experiments were run to evaluate the agent's performance.[3] They typically consisted of two conditions. In the first condition the agent monitors a running simulation but offers no advice. In the second condition the agent monitors a running simulation and implements any advice that it may have. The goal in each of these experiments is to determine how the agent's performance compares to doing nothing. We assess the impact of the agent's actions by performing an analysis of variance (ANOVA) of each cost measure on whether or not agent advice is used. If the result is significant then the actions of the agent affect the cost measure and inspection of cell means will indicate if it is a beneficial effect. A total of ten trials (simulations) were run in each condition. Also, we allowed any type cargo to be carried by any type of ship. Experiments with more constraints are described in Section 2.2.6.

By varying the number of SMRs for a simulation we have some crude control over the frequency and duration of bottlenecks. Few SMRs results in low traffic intensity and few bottlenecks. Many SMRs has the opposite effect. Therefore, we ran an advice vs. no advice experiment at each of three levels of the number of SMRs in the scenario. The results for the 35 SMR case are shown below in Table 1.

Cost	p Value	No Advice Mean	Advice Mean	% Reduction
BP	0.0	232	169.9	26.8
CT	0.0	2659.9	2261.8	15.0
IC	0.0	1542.9	1208.2	21.7
QL	0.0	911.1	598.5	34.3
SD	0.0325	168.2	149	11.4
SU	0.018	181.1	216.1	-19.3

Table 1: Effects of Agent Advice - 35 SMRs

By allowing the agent to follow its own advice, we obtain significant reductions in all cost measures except Ship Utility. In fact, the percent reduction in all cost measures was largest in the most pathological 35 SMR case (compared to the other cases which are not shown).

[3] All experiments and analysis utilized CLASP/CLIP [2].

The agent achieved its design goal of reducing queue length. In doing so, it reduced the amount of time cargo spends sitting idle and actually increased the speed with which cargo travels to its destination locally (decreased Cargo Transit) and globally (decreased Simulated Days).

To investigate the effects of increasing the complexity of the task on the agent's ability to perform, a similar experiment was run with a larger scenario. The number of ports and ships were doubled (to 10 and 40 respectively) and the number of SMRs was set at 60. Ten trials per condition were run. Again, we obtain significant reductions in all cost measures except Ship Utility. Comparing percentage reductions with results from the previous experiments, we found that increasing the complexity of the task increases the extent to which the agent is able to help.

Cost	p Value	No Advice Mean	Advice Mean	% Reduction
BP	0.0	283.5	204.6	27.8
CT	0.0	4284.6	3523.7	17.7
IC	0.0	2222.8	1577.6	29.0
QL	0.0	1406.7	817.4	41.9
SD	0.0	189.2	157.5	16.8
SU	0.5178	320.35	330.6	-3.2

Table 2: Effects of Agent Advice - Large Scenario

The obvious conclusion is that in a wide variety of conditions, the agent is able to reduce the costs associated with a simulation. Neither pathology intensity nor problem complexity seem to nullify its ability to perform. In fact, the agent shines most brightly in precisely those situations where it is needed most, highly pathological and large/complex scenarios.

2.2.5 Control Condition - Random Advice

An experiment was run to determine the effects of demon accuracy on agent performance. The factors that affect demon accuracy (see Section 2.2.2) were varied with the surprising result that there was no significant impact on the efficacy of the agent. If the agent performs equally well with good and bad predictions of bottlenecks, then perhaps its ability to reduce costs is due to shuffling of routes, not to its ability to predict. Random rerouting, periodically picking a piece of cargo and sending it on a newly chosen random route, may be as good as the agent. Random rerouting has the advantage of tending to evenly distribute cargo over the network, minimizing contention for any one port. The disadvantage is that it destroys the structure inherent in the initial schedule.

To investigate the utility of random advice, we ran an experiment in which the agent, with varying frequency, rerouted a randomly selected piece of cargo. This was done with no regard for bottleneck predictions. We varied the probability of performing a reroute on each day over four values: 5%, 15%, 25%, 35%. As with the advice vs. no advice experiments, the number of SMRs in the simulation was varied to get a feel for how these effects changed with pathology intensity. As expected, cost measures decrease with increasing frequency of random rerouting. Random rerouting can be used to improve throughput if one is willing to pay for incremental improvements with additional disruption to the initial schedule.

The most important result of this experiment is that the agent performs better than random rerouting when both throughput and the number of reroutes are considered. Regardless of pathology intensity, there existed a level of random rerouting that matched the performance of the agent when measured in terms of throughput. However, at that level of performance the agent always used significantly fewer reroutes. For a given level of throughput, the agent uses a few well directed rerouting decisions rather than large numbers of random ones, thereby doing less violence to the initial schedule.

The same large scenario described previously was used to investigate the effects of problem size and complexity on the efficacy of random advice. The results here are striking. Increasing the amount of random advice seems to help monotonically. The amount of rerouting performed by the agent was statistically equivalent only to the lowest level of random rerouting. In that condition, the agent's performance is significantly better than random advice; its domain knowledge is paying great rewards. Looking at the data another way, the agent performed equally as well as the highest level of random rerouting with 34% fewer reroutes.

Our initial proposition that random rerouting would help to lower the various cost measures was borne out. In fact, it seems that more random rerouting is better than less, except perhaps in highly bottlenecked scenarios. There existed some level of randomness that equaled the performance of the agent for each of the previous experiments. However, the agent typically rerouted many fewer pieces of cargo than the equivalent level of randomness, thereby preserving more of the structure of the simulation. Finally, it appears that for large/complex scenarios the difference between randomness and the agent is more pronounced.

2.2.6 Highly Constrained Scenarios

To increase the realism of the agent's task, several constraints were added to the scenarios. There are three types of cargo: CONT (containerized), GENL (breakbulk), and RORO (roll on, roll off). Rather than using a single cargo type, we used multiple types and limited the cargo handling capabilities of both ships and docks. Now for cargo to flow through the network, it must match in type with any ship that is to carry it and any dock where it will be handled. We ran agent advice vs. random advice experiments under these conditions after modifying the agent to consider the additional constraints. Whenever random advice generated an incompatible routing assignment, such as sending CONT cargo to a port only equipped to handle GENL, it was penalized with a short delay for the offending piece of cargo. The results are presented below. When a Scheffe' test[4] determines that one of the means at a random level is significantly different from the mean at the agent advice level, an '*' is used to mark the mean at the random level.

Again we see that the agent is able to match the performance of random rerouting with many

[4] A Scheffe' test is a t-test for multiple pairwise comparisons that preserves a specified experimentwise error.

Level	Reroutes	CT	IC	QL
Real Advice	9.6	1833.0	990.9	488.5
Random 5	4.6 *	2010.2 *	1135.6 *	617.4 *
Random 15	16.6 *	1770.4	961.3	534.5
Random 25	22.4 *	1697.0 *	877.9	443.0
Random 35	31.8 *	1651.0 *	860.7 *	452.8

Table 3: Real vs. Random Advice - 30 SMR Constrained Scenario

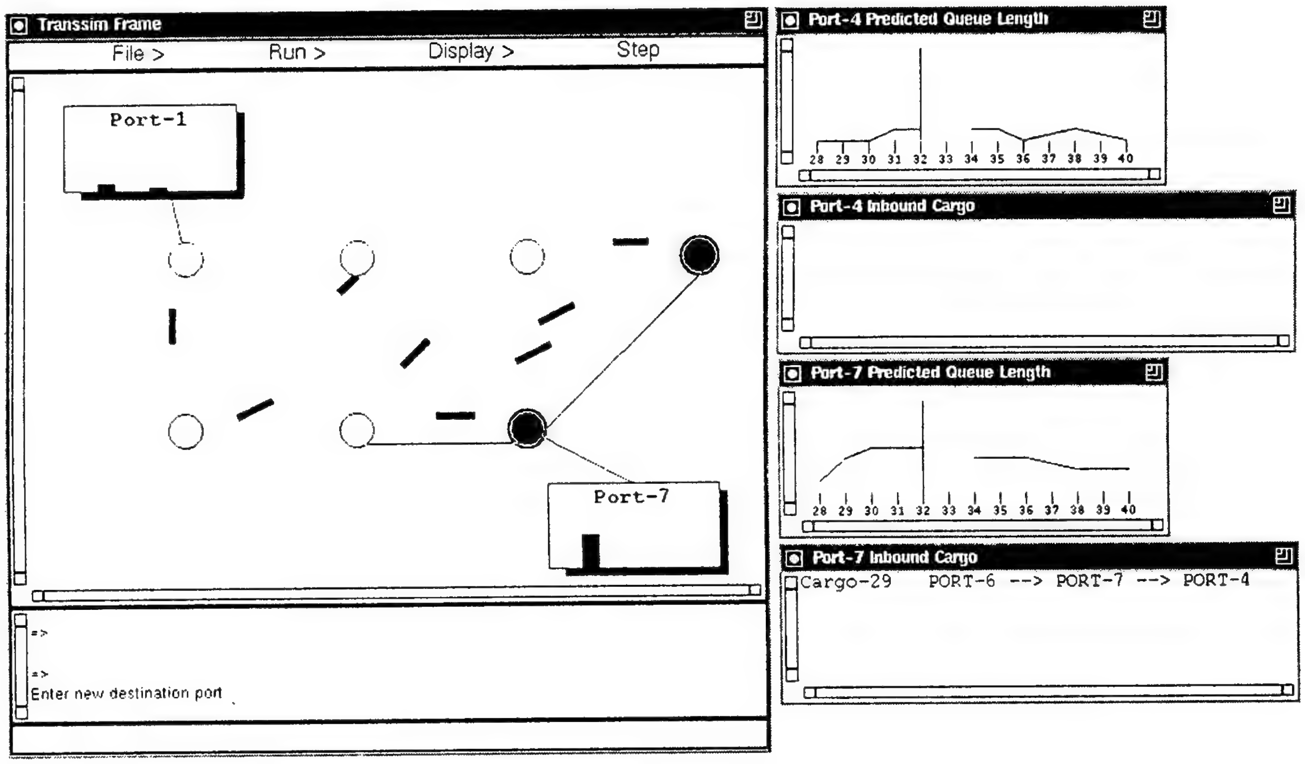
fewer changes to the schedule. The fact that Queue Length for the agent is different only from the Random 5 condition and that Cargo Transit for the agent is different only from the highest and lowest levels of random advice points to a result of constraining the scenario: the performance of random advice in any one condition is highly variable. The variance associated with Cargo Transit for random advice was on average 3.5 times higher than for agent advice. Likewise, the variance associated with Idle Cargo was 4.1 times higher and the variance associated with Queue Length was 3.2 times higher. The agent is able to achieve good average case performance with much higher consistency when compared to random rerouting by making a few appropriate rerouting decisions.

2.3 Bringing Humans into the Loop

Part of the motivation for plan steering is the belief that humans find it extremely difficult to perform tasks such as the one for which our agent was designed. Tracking hundreds of events over time and understanding primary and secondary effects of schedule modifications is not something that people do well. Therefore, we ran a series of experiments in which humans were asked to perform the same task at which the agent was previously evaluated. We provided a set of graphical displays that gave the human user essentially the same information and rerouting capabilities available to the agent. In one half of the trials the human worked alone, and in the other half the human and the agent worked together. This experiment design and experimental results are presented below [6].

2.3.1 Experiment Design

The schedule maintenance agent was designed to increase throughput in TransSim simulations while minimizing schedule disruptions. The goal of this set of experiments is to determine how both an unassisted human and a human working in concert with the agent perform at that task. In each case the human has quick access to roughly the same information and schedule modification capabilities available to the agent. The transportation network is displayed as a connected graph with ship icons moving along the arcs as they traverse simulated channels. The pathology demon's predictions are displayed as a moving graph of queue length vs. simulated day. A sliding window shows both current history of actual queue lengths and predicted queue lengths for several days into the future. One predicted queue length window is on screen for each port during the simulation. Taken together with the network display window, the human user has a simple but informative visualization of the information used by the demon and the agent. A snapshot of the TransSim user interface is shown below in Figure 3.



```
=>
=>
Enter new destination port
```

```
Cargo-29    PORT-6 --> PORT-7 --> PORT-4
```

Figure 3: TRANSSIM User Interface

Just as the agent makes schedule changes by rerouting cargo, so does the human. An inbound cargo window for each port lists the pieces of cargo that are bound for the port but have not yet been loaded into ships and placed in a channel. Cargo routes may be highlighted and modified by simply clicking on a new destination port.

By monitoring the demon's predictions for each port, it is possible to judge the effects of sending additional cargo. If the predicted queue length at a port is low or is trending downward, then it might be a good idea to let cargo continue to be dispatched to that port. If the predicted queue length at a port is high or is trending upward, then it might be a good idea to reroute cargo around the port to a less congested area. The benefit of rerouting is that the cargo in question will not waste time sitting in a long docking queue and the queue at the bottlenecked port will be given time to clear itself out. It is important to remember that the demon's predictions include some error. A predicted bottleneck may never materialize and the rerouting action may have been wasted.

In one half of the trials the human works alone. In the other half the human has the aid of the schedule steering agent. We call these the *unassisted* and *assisted* conditions respectively. In the assisted condition the agent evaluates the state of the network and generates advice for the user. Advice identifies both a port that is thought to be a potential bottleneck and a piece of cargo bound for that port, and suggests an alternative route. The human evaluates the agent's advice via the various displays described earlier and may decide to accept or reject the advice. In either case the human may implement a rerouting decision of their own construction.

Each of the four participants in the experiment ran ten simulations. The first two simulations were training trials to get the user familiar with the task and the available tools. One training trial was assisted and the other unassisted with the order chosen randomly. Next, two groups of 4 trials were presented where all trials in a group were either assisted or unassisted. Again, that ordering was randomized to counterbalance for possible order effects. A total of six different scenarios were used: the first two were always used for training and the remaining four were presented in both the assisted and unassisted conditions. The order of presentation of the scenarios within a grouping was also randomized.

2.3.2 Overall Performance

Several measures of cost were recorded for each simulation (see Section 2.2.3). One-way ANOVA showed a significant main effect of scenario on all of the cost measures. Variance in the structure of the schedules for each of the four scenarios resulted in differing pathology intensities and was ultimately reflected in simulation costs. Therefore, to determine if the presence of agent advice impacted simulation score, we performed two way ANOVA of each cost measure on trial type (assisted or unassisted) and the scenario number. This controlled for variance due to the scenario. There was a significant main effect of trial type on four of the costs: docking queue length, the amount of time that cargo spends sitting idle, the total amount of time that cargo spends in transit, and the number of schedule modifications. All other cost measures were lower in the assisted condition than in the unassisted condition, though they were not significant.

To determine whether agent assistance helped or hurt performance, we used D-tests to compare cost measure means in those conditions.[5] In addition, we let the agent run unhindered on each of the four scenarios, taking its own advice, to see how well it performed. Both assisted and unassisted scores were then compared to means obtained by the agent. The results are presented in the tables below. It is clear from Table 4 that humans working with the help of the agent are able to obtain better throughput than humans working alone. All cost means are lower in the assisted condition although Cargo Transit is not significantly so. Not only does agent assistance result in reduced docking queues and reduced idle time for cargo, but it reduces the amount of time that it takes cargo to travel to its final destination (lower Cargo Transit). This improved performance comes at the expense of disrupting the scenario to a greater extent: on average, about 6 reroutes without assistance, compared to about 12 reroutes with assistance. Since performance is better in the assisted condition, it

[5] A D-test is a randomization two sample t-test that is robust against deviations from parametric assumptions.

is not the case that the agent's advice makes things worse and therefore more intervention is required. Apparently, the agent is bringing pathological states to the attention of the human user that they would otherwise have missed and that the human believes require attention. The agent is serving its intended purpose of helping the human track large numbers of events as they occur in a complex environment.

Cost	**Assisted**	**Unassisted**	**p Value**
Queue Length	742.38	828.19	0.0560
Idle Cargo	1366.56	1497.75	0.0240
Cargo Transit	2750.63	2849.44	0.1420
Reroutes	12.0	6.25	0.0010

Table 4: Comparison of Costs in Assisted vs. Unassisted Trials

How does the human's performance in either condition compare to the agent's? We see in Table 5 that the unassisted human performs significantly worse than the agent in all categories. However, the agent implements almost three times as many changes to the scenario. Neither seems to be striking a good balance between maximizing throughput and minimizing schedule disruption. The results in Table 6 tell a different story. The performance of the assisted human is indistinguishable from the agent's performance; none of the cost measures are significantly different. This result alone is interesting since the agent performs quite well. The difference is that the assisted human is able to achieve this feat with significantly fewer changes to the scenario: 12 reroutes for the assisted human compared to more than 18 for the agent. Apparently our mixed-initiative approach to schedule maintenance is working. As noted before, the agent is probably flagging potential pathologies that the human would have otherwise missed and suggesting schedule modifications. However, the human is selectively filtering the suggestions to implement only those that seem most crucial and that are not wasteful.

Cost	**Unassisted**	**Agent**	**p Value**
Queue Length	828.19	682.88	0.0020
Idle Cargo	1497.75	1272.88	0.0010
Cargo Transit	2849.44	2649.56	0.0150
Reroutes	6.25	18.63	0.0000

Table 5: Comparison of Costs in Unassisted Trials vs. Agent

2.3.3 Evaluating User Decision Points

During an assisted trial, the user is constantly evaluating the state of the network and deciding whether or not to act. We focus on three specific action decisions to determine why the assisted human's performance is so good. They are: the agent offers advice and it is

Cost	Assisted	Agent	p Value
Queue Length	742.38	682.88	0.2220
Idle Cargo	1366.56	1272.88	0.1650
Cargo Transit	2750.63	2649.56	0.2050
Reroutes	12.0	18.63	0.0100

Table 6: Comparison of Costs in Assisted Trials vs. Agent

accepted, the agent offers advice and it is rejected, the human makes a rerouting decision independent of the agent. The problem is one of credit assignment. Is good performance due to the intelligence of the agent? Is it due to the human's ability to differentiate between good and bad advice? Or is it due to the human's ability to formulate schedule modifications independently?

The metric we have chosen for this credit assignment task is daily queue length summed over all ports. Every time during the course of a single simulation that the human makes one of the three decisions, we look at total queue length over a window of fixed size in the future to determine if the decision was good or bad. This is complicated by that fact that there is a heavy trend in queue length. As more and more cargo enters the network, queue lengths grow slowly but steadily to somewhere near the midpoint of the scenario. As cargo leaves the network for final destinations, queue lengths fall off until the scenario ends. The impact of a single user action is easily swamped by the effect of trend. To combat that effect we generated a baseline queue length curve for each of the four scenarios to serve as a standard for expected queue length. That baseline was created by averaging the queue lengths measured for each day over all four of the participants' assisted trials in a scenario and then performing a 3-mean smooth [11]. To score a decision point on a given day in an individual trial, we simply look at future queue lengths in that trial and compare them to future queue lengths in the same time range in the appropriate baseline curve. Subtracting baseline scores from actual scores eliminates trend and gives some idea of performance relative to expected values.

Action	Mean Difference	p Value
Accept Advice	-0.32	0.1790
Reject Advice	0.583	0.0190
User Modification	-1.02	0.0070

Table 7: Decisions Points in Scenario 3

The results for scenario 3 are shown in Table 7. For each action type we computed actual queue length minus expected queue length and compared the mean of those numbers to a mean of zero. In that way we can determine how the actions affect performance over the course of a single simulation when compared to expectation. Accepting the agent's advice results in smaller than expected queue lengths, but the result is not significant. Rejecting the agent's advice led to significantly larger than expected queues. It appears that in this scenario, the agent's advice tends to stave off potential pathologies and ignoring its advice is detrimental. In terms of making beneficial schedule modifications, the human fares quite well. When compared to expectations, the results of the human's rerouting decisions are significantly better. With the tools that we provided, the human was able to evaluate the state of the transportation network, identify potential trouble spots, and formulate a preventative plan. Therefore, poor human performance in the unassisted trials was not due to an inability to understand and manipulate the domain.

2.4 Conclusions and Future Work

We have seen that an intelligent agent may be constructed that takes advantage of minimal domain knowledge and that uses simple heuristics to manage a complex problem domain. Such an agent was constructed for schedule maintenance in a simulated transportation network and its performance was evaluated. The presence of the agent significantly reduced cost measures when compared to doing nothing. In addition, the benefit of the agent grew with problem size and complexity. A default rule for managing the domain with random rerouting was explored and shown to have substantial utility in reducing cost. Unfortunately, random rerouting tended to destroy the structure of the scenarios. At a given level of simulation cost, the agent used fewer rerouting actions than the default rule. As problem size and complexity grew this difference became more pronounced.

Simultaneously achieving the two goals of maximizing throughput and minimizing the number of changes to the initial schedule proved to be difficult for both the human and the agent. The human rerouted few pieces of cargo at the expense of high simulation costs. Experimental results indicate that the humans' individual decisions resulted in significantly better than expected performance. Therefore, poor overall performance by human subjects is not due to their inability to understand the domain. The agent's simulation costs were quite low but the number of cargo rerouting decisions was high. The optimal balance was struck by the agent and the human working together. The agent enhanced the human's ability to identify potential pathologies in a complicated environment, and the human evaluated and filtered away schedule modifications with dubious utility that were suggested by the agent.

The goal of this research is to arrive at a generalizable architecture for plan steering that scales up to the demands of large, complex planning problems. We want to be able to replace TransSim with the real world and have agents working with humans to avoid pathologies in plans and schedules. To that end, we will continue to push on this system by investigating pathologies other than bottlenecks, advice other than rerouting, and methods for increasing predictive accuracy. We want to develop explanatory theories for why our rerouting heuristic works so well in this domain and for why bringing humans into the loop has such a dramatic impact on performance. We then hope to study other problem domains to understand how they are different from transportation planning and how those differences impact the efficacy of our architecture.

Acknowledgments

This research is supported by ARPA/Rome Laboratory under contract #F30602-91-C-0076. The US Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

References

[1] Ambros-Ingerson, J. A. and Steel, S. Integrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 83-88, Minneapolis, Minnesota, 1988.

[2] Anderson, S.D., Carlson, A., Westbrook, D.L., Hart, D.M. and Cohen, P.R. CLASP/CLIP: Common Lisp Analytical Statistics Package/Common Lisp Instrumentation Package. Department of Computer Science Technical Report 93-55, University of Massachusetts, Amherst.

[3] Lopez-Mellado, E. and Alami, R. A failure recovery scheme for assembly workcells. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 1, pages 702-707, 1990.

[4] Muscetolla, N. and Smith, S.F. A probabilistic framework for resource-constrained multi-agent planning. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pp. 1063-1066, Morgan Kaufmann, 1987.

[5] Oates, T. and Cohen, P.R. Toward a plan steering agent: experiments with schedule maintenance. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*. AAAI Press, pp. 134-139. Department of Computer Science Technical Report 94-02, University of Massachusetts, Amherst.

[6] Oates, T. and Cohen, P.R. Humans plus agents maintain schedules better than either alone. Department of Computer Science Technical Report 94-03, University of Massachusetts, Amherst.

[7] Ow, P.S., Smith, S.F. and Thiriez A., Reactive plan revision. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 77-82, Morgan Kaufmann, 1988.

[8] Sadeh, N. Micro-opportunistic scheduling: the micro-boss factory scheduler. To appear in *Intelligent Scheduling*, edited by M. Zweben and M. Fox, Morgan Kaufmann, 1993.

[9] Simmons, R.G. A theory of debugging plans and interpretations. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 94-99, Minneapolis, Minnesota, 1988.

[10] Smith, S.F., Ow, P.S., Muscetolla, N., Potvin, J., and Matthys, D.C. An integrated framework for generating and revising factory schedules. In *Journal of the Operational Research Society*, Vol. 41. No. 6, 1990.

[11] Tukey, J.W. *Exploratory Data Analysis*. Addison-Wesley Publishing Company, 1977.

[12] Wilkins, D.E. Recovering from execution errors in SIPE. Technical Report 346, Artificial Intelligence Center, Computer Science and Technology Center, SRI International, 1985.

3 Detecting Complex Dependencies in Categorical Data

Tim Oates, Dawn E. Gregory, and Paul R. Cohen

Abstract

Locating and evaluating relationships among values in multiple streams of data is a difficult and important task. Consider the data flowing from monitors in an intensive care unit. Readings from various subsets of the monitors are indicative and predictive of certain aspects of the patient's state. We present an algorithm that facilitates discovery and assessment of the strength of such predictive relationships called *Multi-stream Dependency Detection* (MSDD).

We use heuristic search to guide our exploration of the space of potentially interesting dependencies to uncover those that are significant. We begin by reviewing the dependency detection technique described in [3], and extend it to the multiple stream case, describing in detail our heuristic search over the space of possible dependencies. Quantitative evidence for the utility of our approach is provided through a series of experiments with artificially-generated data. In addition, we present results from the application of our algorithm to two real problem domains: feature-based classification and prediction of pathologies in a simulated shipping network.

3.1 Dependency Detection

Consider a network of seaports, with ships carrying cargo between the ports according to a complex schedule. Unforeseen occurrences at a single port or a group of ports, such as severe weather or mechanical failures, can impact the schedule at adjacent ports. If the current state of the network can be used to predict future states of the network, it may be possible to adjust the schedule to minimize adverse effects of such unforeseen occurrences. Similarly, it would be very useful to determine how the future state of a patient, as indicated by various monitors in an Intensive Care Unit (ICU), depends on the current state of the patient. Note that data in the form of time series is not required for the discovery and exploitation of such predictive relationships. Machine learning algorithms that perform feature-based classification determine how a class label depends on various subsets of a feature vector. In all of the above examples, the goal is to determine whether one set of features can be used to predict another set of features. The two sets of features may be taken from the same source at different times (e.g. ICU monitors), or they may be taken from different sources with no notion of time (e.g. a feature vector and a class label). We will revisit both the shipping network and classification examples later in this paper.

A *dependency* is an unexpectedly frequent or infrequent co-occurrence of events over time. Our goal is to find dependencies between tokens contained in multiple streams. A stream is a sequence of values produced over time, and a token is one of the finite set of values that a stream can produce. Dependencies across multiple streams may take many forms: perhaps token A in stream 1 predicts token B in stream 2, or perhaps token A in stream 1 *and* token

Submitted to *Fourteenth International Joint Conference on Artifical Intelligence* (IJCAI-95). Also appeared as shorter version in *Preliminary Papers of the Fifth International Workshop on Artifical Intelligence and Statistics*, pp. 417-423.

C in stream 2 predict token B in stream 2. In general, if stream j contains t_j distinct tokens, there are $[\prod_{j=1}^{n} t_j + 1]^2$ possible dependencies between two items.

The dependency detection technique in [3] uses contingency tables to assess the significance of dependencies in a *single* stream of data. Let (t_p, t_s, δ) denote a dependency. Each dependency *rule* states that when the precursor token, t_p, occurs at time step i in the stream, the successor token, t_s, will occur at time step $i + \delta$ in the stream with some probability. When this probability is high, the dependency is strong.

Consider the stream ACBABACCBAABACBBACBA. Of all 19 pairs of tokens at lag 1 (e.g. AC, CB, BA, ...) 7 pairs have B as the precursor; 6 of those have A as the successor, and one has something other than A (denoted $\overline{\text{A}}$), as the successor. The following contingency table represents this information:

Table(B,A,1) =

	A	$\overline{\text{A}}$	*total*
B	6	1	7
$\overline{\text{B}}$	1	11	12
total	7	12	19

It appears that A depends strongly on B because it almost always follows B and almost never follows anything else ($\overline{\text{B}}$). We can determine the significance of each dependency by computing a G statistic for its contingency table:

$$G\begin{pmatrix} n_1 & n_2 & r_1 \\ n_3 & n_4 & r_2 \\ c_1 & c_2 & t \end{pmatrix} = 2\left[n_1 \log \frac{n_1 t}{r_1 c_1} + n_2 \log \frac{n_2 t}{r_1 c_2} + n_3 \log \frac{n_3 t}{r_2 c_1} + n_4 \log \frac{n_4 t}{r_2 c_2}\right]$$

For example, the contingency table shown above has a G value of 12.38, which is significant at the .001 level, so we reject the null hypothesis that A and B are independent and conclude that (B,A,1) is a real dependency.

We extend this technique to the multiple stream case by introducing the concept of a *multi-token.* A multi-token represents the value of any or all streams at any given time i. For a series with n streams, all multi-tokens will have the form $< x_1, \ldots, x_n >$, where x_j indicates the value in stream j. In order to support the "any or all" requirement, we add a special wildcard symbol, *, to the set of values that may appear in each stream. Thus we can indicate a "don't care" condition by placing an * in the appropriate stream.

For a multi-stream example, consider the following streams:

AC**BABA**CCBAABAC**BA**C**BA**C
BA**CACA**BACBABAB**CA**B**CA**B

The dependency (<B,C>,<A,A>,1) indicated in boldface is significant at the .01 level with a G value of 7.21. The corresponding contingency table is:

Table(B,A,1) =

	<A,A>	$\overline{\text{<A,A>}}$	*total*
<B,C>	4	1	5
$\overline{\text{<B,C>}}$	2	12	14
total	6	13	19

We now have both syntax and semantics for multi-stream dependencies. Syntactically, a dependency can be expressed as a triple containing two multi-tokens (a precursor and a successor) and an integer (the lag). For each of the n streams, the multi-tokens contain either a token that may appear in the stream or a wildcard. Dependencies can also be expressed in the form $x \rightarrow_{\delta} y$ where x and y are multi-tokens. Semantically, this says the occurrence of x indicates or predicts the occurrence of y, δ time steps in the future.

3.2 Searching for Dependencies

The problem of finding significant two-item dependencies can be framed in terms of search. A node in the search space consists of a precursor/successor pair – a predictive rule. The goal is to find predictive rules that are "good" in the sense that they apply often and are accurate. The root of the search space is a pair of multi-tokens with the wildcard in all n positions. For $n = 2$, the root is $< *, * > \rightarrow < *, * >$. The children of a node are generated by replacing (instantiating) a single wildcard in the parent, in either the precursor or successor, with a token that may appear in the appropriate stream. For example, the node $<$ A,* $>\rightarrow<$ *,X $>$ has both $<$ A,Y $>\rightarrow<$ *,X $>$ and $<$ A,* $>\rightarrow<$ B,X $>$ as children.

The rule corresponding to a node is always more specific than the rules of its ancestors and less specific than any of its descendants. This fact can be exploited in the search process by noting that as we move down any branch in the search space, the value in the top left cell of the contingency table (n_1) can only remain the same or get smaller. This leads to a powerful pruning heuristic. Since rules based on infrequently co-occurring pairs of multi-tokens (those with small n_1) are likely to be spurious, we can establish a minimum size for n_1 and prune the search space at any node for which n_1 falls below that cutoff. In practice, this heuristic dramatically reduces the size of the search space that needs to be considered.

Our implementation of the search process makes use of *best first search* with a heuristic evaluation function. That function strikes a tunable balance between the expected number of hits and of false positives for the predictive rules when they are applied to previously unseen data from the same source. We define *aggressiveness* as a parameter, $0 \leq a \leq 1$, that specifies the value assigned to hits relative to the cost associated with false positives. For a given node (rule) and its contingency table, let n_1 be the size of the top left cell, let n_2 be the size of the top right cell, and let t_S be the number of non-wildcards in the successor multi-token. The value assigned to each node in the search space is $S = t_S(an_1 - (1-a)n_2)$. High values of aggressiveness favor large n_1 and thus maximize hits without regard to false positives. Low aggressiveness favors small n_2 and thus minimizes false positives with a potential loss of hits. Since the size of the search space is enormous, we typically impose a limit on the number of nodes expanded. The output of the search is simply a list of the nodes, and thus predictive rules, generated.

3.3 Empirical Evaluation

In this section we evaluate the performance of the algorithm on artificially-generated data sets. The goal is to answer a variety of questions regarding the behavior of the algorithm over its domain of applications. Artificial data simplifies this task since the "real" dependencies are known, providing means for distinguishing structure in the data from noise.

Artificial data sets are generated by random sampling and applying a set of probabilistic *structure rules*: $R = \{(P, Pr_P, S, Pr_S)\}$. Each series is initialized by generating n streams of length l, sampled randomly from the token set T. Values for n, l, T, and R are determined by the experiment protocol. Default values are $n = 5$, $l = 100$, $T = \{\text{A,B,C,D,E}\}$, and $R = \{(<\text{A,A,*,*,*}>, .1, <\text{C,D,D,*,*}>, .8), (<\text{* C,C,*,*}>, .1, <\text{*,A,A,B,*}>, .8), (<\text{*,*,D,D,*}>, .1, <\text{*,*,D,C,B}>, .8)\}$.

Structure is then introduced into this random series in two phases: first, seed the precursors P into each time-slice with probability Pr_P; then, whenever a time-slice i matches the precursor of a rule r, insert the successor into time-slice $i + \delta$ with probability $Pr_S(r)$. For analysis, we can partition the resulting series into noise and structure by determining which components are predicted by the dependency rules $(P(r), S(r), \delta)$ for each structure rule $r \in R$.

In each experiment, we run one or more iterations of the search algorithm for each experiment condition. Unless different values are specified by the experiment protocol, we gather 5000 predictive rules with aggressiveness set to 0.5. These rules are post-processed as described below, and used to make *predictions* in ten new data sets generated from the same structure rules. The results are evaluated with respect to two factors: *predictive power* (the total number of predictions made) and *accuracy* (the percentage of the predictions that were correct). These factors are considered separately for the structure and noise portions of the data set.

3.3.1 Selecting the Best Dependency Rules

The MSDD search algorithm generates a large set of dependencies, from which we would like to select the most accurate and predictive rules. Since all our experiments depend on the quality of this selection process, the first question we wish to answer is, "what post-processing strategy will select the best predictive rules?" Although more sophisticated techniques may be needed to resolve redundancy, the simplest approach is to *filter* and *sort* the rules, first discarding rules that do not conform to certain criteria, and then ranking them according to some precedence function.

In this experiment, four different filter criteria are combined with six different sort functions for a total of 24 experiment conditions. The filter options discard rules under the following conditions: (1) never; (2) G not significant at the 0.05 level ($G < 3.84$); (3) $n_1 < 5$; and (4) $n_1 < n_2$. The remaining rules are then sorted according to one of these six functions: (1) randomly; (2) the G statistic (computed over the training data); (3) the number of true instances n_1; (4) the approximate number of true predictions $n_1 \times t_S$; (5) the percentage of instances that are true $n_1/(n_1+n_2)$; and (6) the approximate percentage of predictions that are true, $n_1 \times t_S/(n_1+n_2)$.

We ran five iterations of each condition on data sets with default structure. The results indicate that the highest predictive power and accuracy are achieved when discarding rules with fewer than 5 true instances (filter condition 3), and sorting them according to the G statistic (sort condition 2). This result is as expected: the rules that remain are unlikely to be spurious dependencies, and they are applied in order of their significance.

3.3.2 Comparison of Search Heuristics

Now that we know how to effectively use the output of MSDD, we can address important issues regarding the performance of the algorithm. In this experiment, we compare the performance of the S heuristic (defined in Section 3.2) to other heuristics and across different levels of aggressiveness.

All the search heuristics used in this experiment are based on contingency table analysis of the dependency rules. In addition to the S heuristic, we also use:

1. A normalized S value $\frac{S}{(n_1+n_2)(n_1+n_3)}$, where S is normalized by its *expected count*.
2. The aggressiveness-weighted ratio of hits to false-positives, $\frac{an_1}{(1-a)n_2}$.
3. The aggressiveness-weighted fraction of the instances that are hits, $\frac{an_1}{n_1+n_2}$.

The results (which are not included here due to space constraints) confirm that S is the best of these heuristics: it produces good accuracy and predictive power while allowing the user to tune the performance with the aggressiveness parameter; the other heuristics are not affected by tuning. As expected, high aggressiveness favors predictive power while low values favor accuracy.

3.3.3 Effects of Inherent Structure

Perhaps the most important question to be resolved is: How strong must a dependency be in order for it to be found by the algorithm? In practical terms, this involves two issues: how frequently a dependency occurs and how often the precursor multitoken appears but the successor multitoken does not. In this experiment, we generated 243 data sets of default size, with 1, 3, or 5 structure rules spanning all combinations of: precursor size $t_P \in \{1,3,5\}$, precursor probability $Pr_P \in \{.1,.2,.3\}$, successor size $t_S \in \{1,3,5\}$, successor probability $Pr_S \in \{.1,.5,.9\}$.

The results of this experiment are very encouraging. They indicate that the successor probability is the only limitation on the accuracy of the algorithm, even though the number of rules, the size and probability of the precursor patterns determine the amount of structure that is available to be predicted. Further exploration is required to confirm these results.

3.3.4 Effects of Problem Size

The final issue to be resolved is the influence of the problem size on the performance of the algorithm. In this experiment, we are primarily concerned with the level of performance attained for a given number of predictive rules as the problem size increases. Ideally, we can bound performance as a polynomial function of the input size.

In this experiment, we generate 27 data sets spanning all combinations of: number of streams $n \in \{5, 10, 20\}$, stream length $l \in \{100, 1000, 5000\}$, and number of tokens $\mid T \mid \in \{5, 10, 20\}$. For each data set, we let MSDD generate 1000, 5000, 10000, and 20000 predictive rules, with aggressiveness set to 0.5.

This experiment has several interesting results. First, performance actually *improves* as the number of tokens increases; intuitively, this is due to the probability of each token decreasing as their numbers increase. Second, the accuracy of the algorithm is basically constant as the stream length increases. This is due to the probability distributions remaining constant as the length increases. The time requirement of the algorithm does increase with stream length. Finally, it appears that MSDD need only generate $n \times 1000$ search nodes to discover the significant dependencies; this is a very strong claim that needs to be supported by further experimentation.

3.4 Applications

Recall two of the example applications from the introduction that were used to motivate our discussion of MSDD: feature-based classification and predicting the state of a shipping network. In this section we discuss the performance of MSDD on both of those tasks.

3.4.1 Feature-Based Classification

In the interest of generality, we applied MSDD to a task for which it was not explicitly designed: feature-based classification. We present results for thirteen datasets from the UC Irvine collection. Twelve of those datasets were selected from a list of thirteen presented in [9] as being a minimal representative set that covers several important features that distinguish problem domains. The precursor multi-tokens were n-ary feature vectors and the successor "multi-tokens" contained only the class label. These pairs of multi-tokens serve as input to the MSDD algorithm. The results are presented below in Table 8. The accuracy shown in the table is the mean obtained over ten trials where the data was randomly split on each trial into a training set containing 2/3 of the instances and a test set containing the remaining 1/3. The exceptions are NetTalk (training data was generated from a list of the 1000 most common English words, and accuracy was tested on the full 20,008 word corpus), Monks-2 (a single trial with 169 training instances and 432 test instances to facilitate comparison with results contained in [7]), Soybean (a single trial with 307 training instances and 376 test instances), and Mushroom (500 training instances and 7624 test instances). We compared MSDD's performance with other published results for each dataset [2, 4, 7, 8]. On ten datasets for which we had multiple published results, MSDD performance exceeds half of the reported results on six datasets. Only on the Soybean dataset did MSDD perform badly. Nearly all of the 20,000 search nodes generated for that dataset were devoted to predicting a single majority class. An unusually large number of highly accurate rules for predicting that class exist, and were therefore found and expanded by the search algorithm. Due to the high branching factor of the Soybean dataset (it contains 35 attributes), the node limit on the search tree was quickly reached. We are currently exploring solutions to this problem. For a more complete comparison than that shown in Table 8, refer to [5].

Data Set	**Mean Accuracy**	**Search Nodes**	**Other Results from Literature**
Breast Cancer	95.15%	10,000	1-nearest neighbor 93.7%
Diabetes	71.33%	10,000	ADAP 76%
Heart Disease	79.21%	20,000	ID3 71.2%; C4 75.8%; back prop 80.6%
Hepatitis	80.77%	10,000	CN2 80.1%; C4 81.2%; Bayes 84.0%
LED-7	70.54%	5,000	CART 71%; C4 72.6%; Bayes 74%
LED-24	71.28%	5,000	CART 70%; NTgrowth+ 71.5%; Bayes 74%
Lymphography	78.16%	15,000	Assistant-86 76%; CN2 82%; Bayes 83%
NetTalk	70.11%	50,000	NetTalk 77%
Monks-2	79.17%	5,000	CN2 69.0%; ID3 69.1%; back prop 100%
Mushroom	99.49%	30,000	GINI 98.6%; Info Gain 98.6%; C4 100%
Soybean	13.83%	20,000	IWN 97.1%
Thyroid	95.46%	20,000	
Waveform-40	73.02%	15,000	Nearest neighbor 38%; CART 72%; Bayes 86%

Table 8: Performance of MSDD as a feature-based classifier on thirteen datasets from the UC Irvine collection.

3.4.2 Pathology Prediction

We applied MSDD to the task of predicting pathologies in a simulated shipping network called TransSim. When several ships attempt to dock at a single port at the same time, most will be queued to await a free dock, resulting in a *bottleneck*. We built a pathology demon that predicts the potential for bottlenecks before they actually form, and we built an agent that modifies the shipping schedule in an effort to keep predicted pathologies from materializing. Using the demon as an oracle, we gathered data from a single run of the simulator and used MSDD to generate rules to predict bottlenecks. To assess the utility of the previously generated rules, we ran ten simulations in each of two conditions; one with the existing demon and another with the demon replaced by the rules. We used t tests to determine whether mean costs associated with each simulation were lower in the demon condition as compared to the rule condition. The results are presented below in Table 9. Note that the number of pathologies predicted (PP) by the demon is almost twice the number predicted by the rules and, therefore, the agent made about twice as many schedule modifications (SM). However, of the five cost measures (QL, IC, CT, SU, and SD) only SD was significantly lower in the demon condition when compared to the rule condition. That is, even though the agent is taking a much more active role, performance is not significantly better. Inspection of execution traces shows that the demon is much more likely than the rule set to predict short-lived pathologies. The rules are good at forecasting substantial pathologies, ones that will not go away of their own accord, but miss the more fleeting pathologies. Said differently, MSDD rules are not misled by small, noisy fluctuations in the state of the simulation. This behavior is beneficial when we view disruption to the original schedule as a cost that we want to minimize.

Cost	Demon Mean	Rule Mean	p Value
PP	184.2	94.6	0.0001
CT	2289.3	2377.9	0.0689
IC	1149.8	1202.1	0.1844
QL	637.7	640.5	0.9177
SD	131.1	141.6	0.0019
SU	188.8	202.2	0.3475
SM	21.6	9.2	0.0001

Table 9: Comparison of simulation costs using demon and MSDD rules for pathology prediction.

This experiment suggests that MSDD can discover indicators of pathological states in TransSim from high level domain information. MSDD can identify relevant state information to emulate the objective function of an external oracle. One limitation of this approach, as compared with the demon, is that an initial run of the simulator is required to gather data to drive the rule generation process. However, the domain knowledge supplied to the MSDD algorithm was minimal in comparison to the demon.

3.5 Conclusion

In this paper we described how the problem of finding significant dependencies between the tokens in multiple streams of data can be framed in terms of search. The notion of dependencies between pairs of tokens introduced in [3] was extended to pairs of multi-tokens, where a multi-token describes the contents of several streams rather than just one. We introduced the Multi-Stream Dependency Detection (MSDD) algorithm that performs a general-to-specific best-first search over the exponentially sized space of possible dependencies between multi-tokens. The search heuristic employed by MSDD strikes a tunable balance between the expected number of hits and false positives for the dependencies discovered when they are applied as predictive rules to previously unseen data from the same source. We presented results from an empirical evaluation of MSDD's performance over a wide range of artificially generated data. In addition, we applied MSDD to the task of pathology prediction in a simulated shipping network and to a number of classification problems from the UC Irvine collection. The results that we obtained are very encouraging.

We are currently working on an incremental version of MSDD that can identify dependencies by processing data as it is generated, and that adapts to changing probability distributions. Also, we are working to remove the need for a fixed sized multi-token and a fixed time interval between multi-tokens.

Acknowledgments

This work is supported by ARPA/Rome Laboratory under contract #'s F30602-91-C-0076 and F30602-93-C-0010. The U.S. government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

The heart disease data was gathered by Andras Janosi, M.D. at the Hungarian Institute of Cardiology, Budapest; William Steinbrunn, M.D. at the University Hospital, Zurich, Switzerland; Matthias Pfisterer, M.D. at the University Hospital, Basel, Switzerland; Robert Detrano, M.D., Ph.D., V.A. Medical Center, Long Beach and Cleveland Clinic Foundation. The lymphography data was obtained from the University Medical Centre, Institute of Oncology, Ljubljana, Yugoslavia from M. Zwitter and M. Soklic. The breast cancer data was obtained from the University of Wisconsin Hospitals, Madison from Dr. William H. Wolberg [1].

References

[1] Bennett, K. P. and Mangasarian, O. L. Robust linear programming discrimination of two linearly inseparable sets. In *Optimization Methods and Software* 1, 1992, 23-34 (Gordon and Breach Science Publishers).

[2] Holte, Robert C. Very simple classification rules perform well on most commonly used datasets. In *Machine Learning*, (11), pp. 63-91, 1993.

[3] Howe, Adele E. and Cohen, Paul R. Understanding Planner Behavior. To appear in *AI Journal*, Winter 1995.

[4] Murphy, P. M., and Aha, D. W. *UCI Repository of machine learning databases* [Machine-readable data repository]. Irvine, CA: University of California, Department of Information and Computer Science, 1994.

[5] Oates, Tim. MSDD as a Tool for Classification. Memo 94-29, Experimental Knowledge Systems Laboratory, Department of Computer Science, University of Massachusetts, Amherst, 1994. Available at http://eksl-www.cs.umass.edu/papers/msdd-classification.ps.

[6] Oates, Tim and Cohen, Paul R. Toward a plan steering agent: Experiments with schedule maintenance. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pp. 134-139, 1994. Available at ftp://ftp.cs.umass.edu/pub/eksl/tech-reports/94-02.ps.

[7] Thrun, S.B. The MONK's problems: A performance comparison of different learning algorithms. Carnegie Mellon University, CMU-CS-91-197, 1991.

[8] Wirth, J. and Catlett, J. Experiments on the costs and benefits of windowing in ID3. In *Proceedings of the Fifth International Conference on Machine Learning*, pp. 87-99, 1988.

[9] Zheng, Zijian. A benchmark for classifier learning. Basser Department of Computer Science, University of Sydney, NSW.

4 Tools for Experiments in Planning

Scott D. Anderson, Adam Carlson, David Westbrook, David M. Hart and Paul R. Cohen

Abstract

The paper describes two separate but synergistic tools for running experiments on large Lisp systems such as Artificial Intelligence planning systems, by which we mean systems that produce plans and execute them in some kind of simulator. The first tool, called CLIP (Common Lisp Instrumentation Package), allows the researcher to define and run experiments, including experimental conditions (parameter values of the planner or simulator) and data to be collected. The data are written out to data files that can be analyzed by statistics software. The second tool, called CLASP (Common Lisp Analytical Statistics Package), allows the researcher to analyze data from experiments by using graphics, statistical tests, and various kinds of data manipulation. CLASP has a graphical user interface (using CLIM, the Common Lisp Interface Manager) and also allows data to be directly processed by Lisp functions.

4.1 Introduction

As planning problems become more complex, involving hundreds of objects and thousands of resources (e.g., ships, planes, tanks, satellites), researchers will need to turn to simulators, controlled experiments, and statistics to study the behavior of their systems. We will briefly describe one such simulator, called TRANSSIM, and a controlled experiment that the Experimental Knowledge Systems Laboratory (EKSL) ran using it, but our real purpose in this description is to introduce two tools that EKSL has developed to aid in running and analyzing experiments of this sort: CLIP and CLASP (Common Lisp Instrumentation Package and Common Lisp Analytical Statistics Package).

CLIP enables researchers to define experiments in terms of the conditions under which the simulator is to be run and the data to be collected. CLIP also helps with the running of the experiment, by looping over all the experimental conditions, running the simulator, and writing the data to files. At that point, a researcher will want to analyze the data using statistical software. While the data files that CLIP writes can be analyzed by any statistical package, CLIP is especially well integrated with CLASP, which is a statistical package that EKSL has implemented. CLASP has many of the standard descriptive and inferential statistics, together with a convenient graphical user interface, and a Lisp interaction window that researchers can use for doing statistical operations that we have not anticipated.

This paper describes TRANSSIM, CLIP and CLASP, and presents an example of their use in an experiment, from initial description to final analysis.

In *Proceedings of the Sixth International IEEE Conference on Tools with Artificial Intelligence* (1994). IEEE Computer Society Press, pp. 615-623.

4.2 Transportation simulator

TRANSSIM simulates the execution of transportation plans in a problem domain where the goal is to get cargo through a shipping network from a number of starting locations to a number of destination ports. The problem, defined as part of the ARPA/RL Planning Initiative, involves many different kinds of cargo and ships, and many, many pieces of cargo. TRANSSIM allows the user to configure an arbitrary shipping network by specifying ports, a set of cargo inputs, and a list of available ships. Input to the scenario is a list of Simple Movement Requirements (SMRs). SMRs specify a port of embarkation, various intermediate ports, and a port of debarkation. Cargo appears at its port of embarkation at times determined by the scenario and travels through the network along the route specified by its SMR. The time for a ship to travel between ports is a Gaussian random variable computed by the simulator and controlled by user-specifiable parameters giving the mean and variance of the ship's speed.

TRANSSIM also supports Interactive Plan Steering, where a human user or a software agent notices problems (pathologies) in the execution of a plan and intervenes in an attempt to get the plan back on course. Currently, our Plan Steering Agent works without reference to a plan or schedule. It attempts to control the shipping traffic by using limited look-ahead for prediction, and it reacts to pathologies as they are detected. One important kind of pathology is when the number of ships arriving at a port exceeds the capacity of the port (the number of docks), so that the ships must wait until docks are available before they can unload.

We have developed a "pathology demon" to try to predict this pathology. Its prediction is for a specified number of days in the future, say four days. The demon looks at each ship heading for a particular port and uses the mean and variance on the ship's speed to estimate the probability that the ship will arrive on the day in question. If that probability exceeds some threshold, the demon assumes that the ship arrives. The demon also predicts how many ships will leave the port, using heuristic estimates about the time it takes to unload and load a ship. All this information is compiled into an estimate of how many ships will be in port on the day in question. If the estimate is higher than the port capacity, the pathology demon can alert the Plan Steering Agent (who may be human); the Plan Steering Agent can then decide what to do, which might include re-routing some of the ships or ignoring the problem because of global considerations. The pathology, of course, is only a local problem, and may be no great hindrance to the overall plan. To study the extent to which local problems affect plan performance, or whether the pathology demon is good at predicting the number of ships in port, or any of myriad other questions, we will need to run experiments, collect statistics, and analyze them. To do that, we will use CLIP and CLASP.

4.3 Running experiments

A great many experiment designs are used in science, but most of them can be viewed as sets of *trials*, each with a number of independent variables, representing the conditions under which the trial is run, and a number of dependent variables, which are the objects of scientific scrutiny. This is the simplest of the kinds of experiment designs that CLIP supports.

One common kind of experiment within this paradigm is called a "fully factorial" design, in which there are one or more *factors*, each of which has a small number of discrete levels. For example, factor A might be the number of days in advance that the pathology demon tries to predict the number of ships in port, with three values (levels)—2, 4 and 6 days. Factor B might be the probability threshold, above which the demon assumes that the ship will be in port, say with levels 0.25, 0.5 and 0.75. A fully factorial experiment design will test all combinations of levels; in this example, there are nine experiment conditions. Because of random variance in the outcome of each trial, the experimenter will usually want multiple trials in each condition and will probably analyze the data using the statistical technique of Analysis of Variance. It's easy to do this kind of experiment using CLIP and CLASP: we tell CLIP how to modify the parameters of the pathology demon and it takes care of iterating through all the conditions, setting the parameters, and collecting the data. Later, CLASP can analyze the data, using just a few mouse clicks, since the Analysis of Variance is built in.

Another common kind of experiment looks at the relationship of two or more continuous variables, such as the correlation between them. For example, the independent variables (variables controlled by the experimenter) might be the number of cargo units to be shipped and the amount of variance in ship speed, while the dependent variable (a variable measured by the experimenter) might be the amount that the plan is late or the number of missed deadlines. We expect that as the scenario becomes more difficult (when the values of the independent variables increase), the plan lateness and missed deadlines will go up—but will this relationship be linear or non-linear? To answer such questions, we will want to run many trials, choosing values for the continuous independent variables and measuring the dependent variables. CLIP can help us do this, while CLASP can graphically display the data and transformations of it, together with regression lines, if desired.

4.3.1 Instrumentation

Adding code to extract information from a system is called *instrumentation*, hence CLIP's name. Most of CLIP's functionality is directed towards extracting different kinds of information from your system—information that is calculated afterwards, collected periodically during execution, or possibly collected whenever some event occurs. This aspect of CLIP is deferred to section 4.3.2. First, we present an overview of how CLIP works and what you need to do to use it. (This article is no substitute for the CLIP/CLASP manual [1], where everything is rigorously explained.)

To use CLIP to run an experiment, CLIP first needs to know how to run your simulator. Essentially, this is a single function or piece of code that CLIP can call to start a trial and which will return when the trial is over. CLIP also works with simulators that run in multi-threaded (multiple process) Lisps, but it nevertheless treats the simulator as a single piece of code. (This requirement may be lifted in future versions of CLIP, but the impact is minor. Most multi-threaded Lisps provide a `process-wait` function, which can be used to make the simulator seem like a single piece of code.) Between trials, CLIP will need to reset your system, although this might be unnecessary if the simulator is purely functional (few are). If your simulator has a notion of time, such as having a clock, and you want CLIP to schedule events for particular times, CLIP will need to know how to interact with the scheduler and the clock. For example, you might want to collect data every day of the simulation, with the average being written to the data file. To describe how to run and control your simulator, there is a single CLIP macro, called `define-simulator`.

Next, you will define your experiment, which is again done with a single CLIP macro, called **`define-experiment`**. The heart of an experiment is the set of independent and dependent variables, which are specified with the macro. The independent variables are described with a simple syntax much like the Common Lisp `loop` macro. The names of dependent variables are simply listed—how to collect and report the data for each dependent variable is separately defined via objects called "clips," which will be discussed in the next subsection. The **`define-experiment`** macro also provides ways for the user to run code of their own choosing during the experiment, at four distinct times:

Before the Experiment: When an experiment gets started, you may want, for example, to load special knowledge-bases or set scenario parameters. This is also a chance to do more mundane things, such as allocating data structures or turning off the screen-saver.

Before Each Trial: At each trial, you may want to reinitialize parameters and data structures. One important thing to do is to configure your simulator for the current experimental condition. For example, if you are running a two-factor experiment, CLIP will have two local variables bound to the correct values of those two factors. You may then use those variables to, perhaps, set parameters of your simulator or use them as arguments to initialization functions. After all, only you know the semantics of your factors.

After Each Trial: The most important thing that is typically done after each trial is to call the function **`write-current-experiment-data`**, the CLIP function that writes all the data for this trial. This is also a good time to run the garbage collector, if you want to minimize garbage collection during trials.

After the Experiment: Typically, code run after the experiment undoes the code run before the experiment, such as deleting data structures or turning on the screen saver.

Of course, any arbitrary code can be executed at these times, for whatever purposes you want. The key idea is that the before- and after-trial code surrounds every trial and runs many times, while the before- and after-experiment code surrounds the whole experiment and runs only once. This ability to run arbitrary code is more than just an opportunity for hacks—it is a clear and precise record of the exact experimental conditions. Records are important as a memory aid and as a means for replicating experiments.

When the experiment has been defined, you start it running with the function `run-experiment`. This function takes arguments, which you can refer to in the before/after code, so that the final specification of the experimental conditions can be deferred until run-time. (For the sake of record-keeping, these arguments should be written to the data file, by using the CLIP function **`append-extra-header`** in the before-experiment code.) The **`run-experiment`** function also allows you to specify the output file for the data, the number of trials, the length of the trial, and other such information.

Defining the simulator and the experiment, and then running the experiment is fairly straightforward and is only a fraction of what must be done to run an experiment. The bulk of the effort is in defining "clips"—functions that measure the dependent variables of your experiment. Fortunately, they are modular and reusable.

4.3.2 Clips

Clips are named by analogy with the "alligator clips" that connect diagnostic meters to electrical devices. They measure and record aspects of your system (the values need not be numerical). Essentially, they are Lisp functions that you define and which CLIP runs if they are included in the definition of the experiment. Once written, they can be mentioned in any number of experiments. Indeed, it's common to build up files of clips, so that a new experiment can be quickly defined by writing a **`define-experiment`** form (or editing an old one) and listing the clips in the **`instrumentation`** argument to **`define-experiment`**.

Clips are defined with the **`defclip`** macro, which is very much like **`defun`**, except that information added before the body is read by CLIP. The central issue in defining a clip is the time that it is run. (The code that is run is written in the **`defclip`** body and is entirely up to the user.) Most clips simply measure values after a trial is finished, for variables such as "finish date," "number of bottlenecks," and "total waiting time for ships." More complicated clips may need to run periodically, which only makes sense for simulators that have a clock of some sort; CLIP will schedule the clip using the **`schedule-function`** specified in the **`define-simulator`** form. Other clips may need to run when some event occurs; this is accomplished by tying the clip to a function in your simulator, using a mechanism like the "advise" facility found in many Lisp implementations. The **`defclip`** form has syntax for tying the clip to a function. When a clip is run many times during a trial, it can either report the mean of the values or it can report all the values (or some function of them), as *time series* data (see section 4.3.3).

CLIP implements several features to make clips more useful and powerful. The first feature allows a clip to report several values to the data file. In other words, if we think of the data file as a large table, with a row for every trial and a column for each variable, a clip may report the values for several columns. For example, a clip that interrogates a port might want to report the minimum, maximum, and mean queue length. The user can define a clip called `queue-info` to report all three of these values during an experiment. The second feature allows users to report a value for each of several objects. For example, they might want to report the maximum queue length at each port, or the tons of cargo carried by each ship. Given a clip to report the value for a single object, another clip can be defined that maps over the objects, calling the simpler clip for each object. These two features can be combined, yielding one clip that reports a lot of information about many objects, all in one powerful step. An important restriction is that the number of values must be consistent, because the data files need to have the same number of values (columns) reported for every trial. This is not a requirement of CLIP so much as a requirement of the statistical package, whether CLASP or any other package. Missing values are a headache for any statistical operation, and so it is better to always produce the same number of values. Typically, this is easy to accomplish. For example, the number of ports should be the same in every trial; if they are not, you will probably be comparing average behavior (since you cannot compare them pairwise), in which case the average can be reported, rather than data for each port.

4.3.3 Time series data

So far, we have described different kinds of data that can be extracted into a "snapshot" of the scenario. We can also collect data that is a "movie" of the system: a series of snapshots at different points in time. Data like this is called *time series* data. For example, we could report the queue length at a port each day, allowing us to see bottlenecks arise and subside as the traffic ebbs and flows. We can statistically analyze such data to see if there are temporal correlations. For example, we could see whether a bottleneck truly subsides or merely moves to another port at a later time. We just cannot answer such questions by looking at mean values after a trial is over.

One trouble with time series data is that it is incompatible with the data collected after the trial. Different kinds of data are collected by time series clips: individual values during a trial versus means and totals afterwards. Usually, a different number of values are produced. It doesn't make sense to mix the two. Therefore, time series data are written to a different file than the main data file. In fact, you can collect several different kinds of time series data in one experiment. For example, you can collect information on port queues every day and collect information every time a ship is loaded. Again because of incompatibility of the data, these two different time series would be written to different files. Someone with such a complex experiment often makes a directory into which all of the data files will go.

As with simple end-of-trial clips, clips for time series can be scheduled to run periodically, as with our once-a-day collection of information on queues, or can be triggered by events, as with our collection of information whenever a ship is loaded. The syntax of `defclip` makes all this relatively easy.

4.3.4 Summary

The capabilities of CLIP have been driven by the needs of experimenters. There are a great many features, all of which have proven useful to someone. Nevertheless, the essence is fairly straightforward. To run an experiment using CLIP, you must do the following:

Define the Simulator You tell CLIP how to run your system via the `define-simulator` macro.

Define the Clips You define a bunch of functions to report the data you want to collect. Most clips will simply return one value, which CLIP will write to the main data file, but you can also define clips that return multiple values, map over multiple objects, run multiple times, or any combination of these features.

Define the Experiment You finally put all of the pieces together by specifying the simulator to run, the experimental conditions (particularly the independent variables), special initialization/parameterization code, and the list of data values to be collected. A call to `write-current-experiment-data` is put here.

Run the Experiment Very little else needs to be specified when the experiment is finally run; usually only the output file, possibly the number of repetitions, and maybe one or two arguments that are referred to in the user's experiment code (`:before-experiment` and the other clauses).

CLIP has other features to support experimentation, such as aborting a trial but continuing the experiment, say when some intermittent error has occurred—very common in stochastic simulations. CLIP also lets you run only part of the experiment, which facilitates breaking the experiment into parts to run on different machines. These are all explained at length in the CLIP/CLASP documentation [1].

An alternative to CLIP is the METERS system, developed by Bolt, Beranek and Newman, Inc., for use in the ARPA/RL Planning Initiative's Common Prototyping Environment (CPE) [2]. METERS is particularly useful for collecting and filtering time-series data from distributed systems.

4.4 Data analysis

The idea of CLASP began when we wanted to run a *t*-test on some experiment data without having to write out the data to a file in some tab-delimited format, move the code to another machine, run a statistics program, and load the data. From this small beginning, we have added most of the workhorse statistical functions, data manipulation (regrouping, selecting subsets), data transformation (such as log transforms), graphing software (now replaced by SCIGRAPH, by Bolt, Beranek and Newman, Inc.). We have a convenient graphical user interface implemented in CLIM, and a programmatic interface so that the CLASP functions can be called by the user if the desired data manipulation isn't already on a menu. Ideally, everything can be accomplished by menus in the graphical user interface.

CLASP's screen interface, an example of which is shown later in figure 4, comprises four areas: the menus, the datasets, the results, and the notebook:

Menus The CLASP menus will appear across the top of the window. The menus, which will be discussed below, are: File, Graph, Describe, Test, Manipulate, Transform and Sample.

Datasets When you load a file of data into CLASP, such as a file written by CLIP, it becomes a CLASP *dataset* and appears on this menu. The name of the dataset is the name of the experiment. Each column of data is called a *variable*; the name of the variable is usually the name of the clip that returned that variable, unless you specify a different name in the `defclip`. When analyzing the main data file (as opposed to a file of time series data), there will be as many variables as there were clip values, and each variable will have as many elements as there were trials, since each clip reports once at the end of each trial. (CLIP has a naming scheme to handle clips that produce multiple values.)

Most operations in CLASP take either datasets or variables as arguments, and the items in this pane become mouse-sensitive under those circumstances. For example, if you want to find the mean number of days cargo spends in transit (and you had a clip that reported that value), you would just select the "Mean" item from the "Describe" menu, whereupon all the variables would become mouse sensitive, and you could select the one you want. Similarly, when you want to partition a dataset, say to separate trials where the Plan Steering Agent was used from those where it wasn't, you would first selecte the "partition" command from the "Manipulate" menu item, and then click on your dataset.

Results Display When a CLASP operation yields a complex result, such as a table or graph, that object goes into a menu of results. The most common use for this menu is to bring up two results side-by-side, so they can be compared. Graphs can often be overlaid, so that similarities are obvious. There are also CLASP commands to delete, print, display, and otherwise operate on results, whereupon they become mouse sensitive.

Notebook The notebook is by far the largest part of the CLASP window because most of the action goes on here. It is a complete Lisp read-eval-print loop, except that CLASP commands are also accepted. Having Lisp available is important and powerful, because users can operate on the data in ways we have not yet implemented or even thought of.

CLASP commands can be typed instead of using the menus; indeed the menus just type the appropriate thing into the notebook. When the command is fully entered, it's executed and its results are printed to the notebook. CLASP output in the notebook is also mouse-sensitive when appropriate.

One of the nice features of the notebook is that it provides a record of the statistical operations on the data. This record can be saved to a POSTSCRIPT® file and printed.

CLASP uses a prefix command syntax, very much like Lisp, in that you give the command name first, such as `:T Test Two Samples` X Y, where X and Y are variables. Using the features of CLIM, CLASP allows command completion and prompts for arguments. CLASP also allows certain arguments to be "mapped," which means that when a list of arguments is given where one is expected, the command is executed for each element of that list. For example, to find the means of three variables, (`X Y Z`), you can use the following syntax:

```
:Mean X,Y,Z
```

CLASP groups related commands in the main menu, as given below. The description just summarizes the kinds of commands; full information is in the CLIP/CLASP manual.

File This menu allows you to load CLASP datasets from files and to save them to files, say if you've made changes or created new datasets. It also allows you to read and write datasets in formats understood by other statistical packages. A number of other utilities are on this menu, such as printing objects (such as graphs or tables) to POSTSCRIPT files. (Currently, CLASP uses CLIM 1.1, which does not produce encapsulated POSTSCRIPT (EPS). We will soon complete an implementation using CLIM 2.0, which will produce EPS, making it easy to insert graphs into other documents.)

Graph Being able to look at your data in various ways is important in exploratory analysis. You may find discontinuous or skewed distributions, non-linearities in trend, or peculiar clusters of data. Looking at the data will suggest new hypotheses and statistical operations, such as smoothing or correlation. This menu allows a number of displays of data, including histograms, scatter plots, line plots, and regression plots. The grapher, BBN's SCIGRAPH, allows graphs to be overlaid for ease of comparison. It also allows the objects (points or lines) in a plot to be colored based on some other property, another important tool for exploratory data analysis.

Describe Statistics are often divided into descriptive statistics and inferential statistics. The former are functions that capture some property of one or more samples, such as location (mean, median), spread (variance, interquartile range) or other properties (correlation between two variables). The latter are functions that test hypotheses about the populations that the samples were drawn from. This menu contains many of the descriptive statistics, including all the ones just mentioned, and a few others, such as modes, trimmed means, arbitrary quantiles, cross-correlations, and auto-correlations. There is also a "statistical summary" operation that prints most of the interesting one-sample statistics in one convenient table.

Test This menu contains the inferential statistics that were omitted from the previous menu. Most of these commands, such as the *t*-test, confidence intervals, Analysis of Variance, Chi-square and Regression, are described in any statistics textbook. One other, called the *d*-test may be unfamiliar, since it is a bootstrap statistic to compete with the *t*-test. Bootstrap statistics [3, 4] replace the parametric and distributional assumptions of statistics like the *t*-test with an empirical approach using computerized resampling of the data. The *d*-test is used just like the *t*-test, especially when the data don't satisfy the normality and equal-variance assumptions of the *t*-test. In the near future, we hope to implement bootstrap variants on most common statistical functions.

Manipulate An experiment usually produces lots of data, which must be broken into pieces to be looked at and understood. Therefore, CLASP provides several ways to extract subsets from a dataset. One example is partitioning, where you select a dataset and a categorial variable from that data. A categorial variable has a few discrete values: for instance, in the shipping domain used in TRANSSIM, the variable `ship-type` might have discrete values like `container`, `tanker`, and `Roll-on/Roll-off`. The partition operation produces new datasets (which appear in the dataset window), one for each distinct value of the categorial variable. You can then select one of these datasets if you want to look just at one value of the variable, say, the "Roll-on/Roll-off" data. Similar operations allow you to partition datasets by an arbitrary predicate (one that you type in).

Other operations on this menu allow you to create new datasets. The values for these new datasets may be cobbled together from existing datasets or come from Lisp functions you executed in the notebook.

When new datasets are produced, whether by partitioning or other operations, a new name is generated, by combining the old name with the operation. This means you can often remember what a dataset is just by looking at its generated name. For example, a dataset `SHIPS` that has been partitioned by its `TYPE` variable, which has a `TANKER` value, will result in a new dataset named `SHIPS (TYPE = TANKER)`.

Transform This menu has commands that produce new variables from old ones. A trivial example is just to sort the variable. A more interesting one is a logarithmic transformation that might be used prior to linear regression, resulting in an exponential model of the data. Another example is smoothing the data, which might be used prior to autocorrelation in order to find cyclical patterns in time-series data. As with datasets, when a new variable is produced, a new name is generated by combining the old name with the operation. For example, a variable named `QUEUE-LENGTH` that has been smoothed will result in a new variable named `SMOOTH-OF-QUEUE-LENGTH`.

Sample This menu contains commands that produce artificial data by sampling from a given probability distribution. These commands would rarely be used in ordinary data analysis, but they are pedagogically important, to see how various graphing options and statistical tests work on data with known properties. The commands can draw numbers from the uniform, normal, binomial, Poisson, and gamma distributions.

4.5 Example

Rather than try to describe in detail how CLIP and CLASP work, we will present an example of using them to run and analyze an experiment in the Transportation Planning domain. The example uses the TRANSSIM simulator and is based on a pilot experiment that Tim Oates used in designing his Plan Steering Agent [5, 6, 7]. The purpose of the experiment is to assess the error rate of a demon that predicts the queue length at a port *d* days in advance, as a function of the variability of ship speed and the time delay, *d*.

The following defines the TRANSSIM simulator. It's quite simple because we won't be using any time-series collection in this experiment.

```
(clip:define-simulator transsim
  :system-name  "TransSim"
  :start-system (simulate nil)
  :reset-system initialize-simulation)
```

Our example experiment will measure the accuracy of the demon that predicts queue lengths at ports and is defined at the top of the next page. Its `:instrumentation` clause mentions three clips for the dependent variables: in this experiment, we are interested in the error rate of the demons in predicting queue length, and in their misses and false positives in predicting bottlenecks. The `:ivs` clause specifies two independent variables—the variance in ship arrival time and the number of days in the future to predict the queue length. In this experiment, the only thing to do before each trial is to transfer the values of the independent variables to the appropriate global variables of the simulator. After each trial, the trial number and the values of independent variables and the clips are written to the data file.

```
(clip:define-experiment pred-accuracy ()
  :simulator          transsim
  :instrumentation    (err-rate fp misses)
  :ivs                ((eta-var in '(0.05 0.15 0.25))
                       (pred-pt in '(2 4 6)))
  :before-trial       (setf *eta-variance-multiplier*
                             eta-var
                            *prediction-points*
                            (list pred-pt))
  :after-trial        (write-current-experiment-data))
```

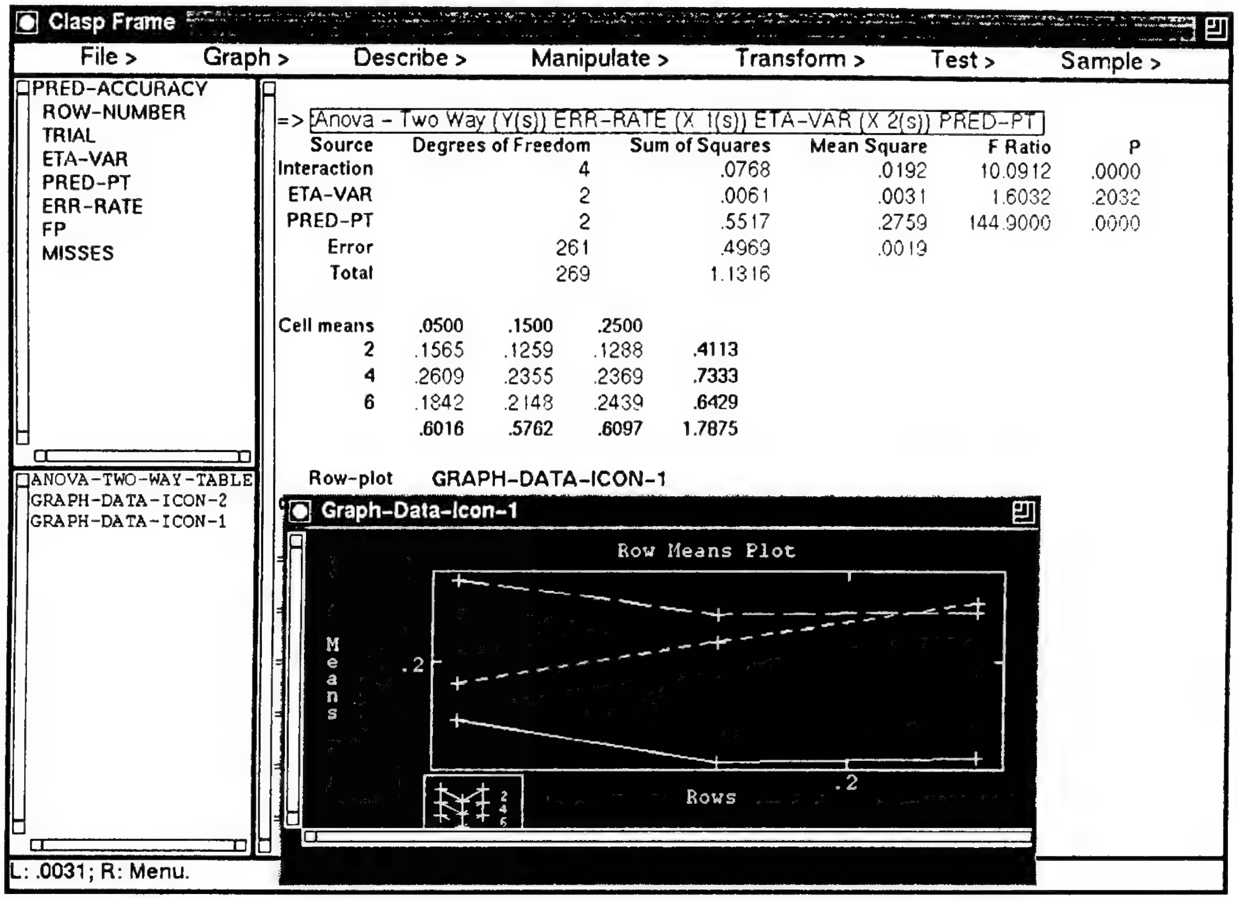
Below is the code for one of the clips in the experiment. It looks just like a Lisp **defun**, except for the **()** before the code. That list is used for specifying additional information such as whether this is a time series clip (by default, clips are not time series), whether it maps over several objects, and so forth. The information is specified in property-list style. Since each port has its own prediction demon, this clip reports the mean error rate over all the demons.

```
(defclip err-rate () ()
  (loop for p in *ports*
        sum (demon-error-rate (port-demon p))
            into total
        finally (return (/ total (length *ports*)))))
```

The experiment is run by executing the following code. The `:repetitions` clause says how many trials to run under each condition (combination of levels of the independent variables).

```
(run-experiment 'pred-accuracy
    :output-file "~/data/demon-summary.clasp"
    :repetitions 30)
```

When the experiment is complete, we will want to analyze the data using CLASP. We are interested in whether either independent variable affects the demon's error rate, and, if so, whether those effects interact. Therefore, we will analyze the data with a two-way Analysis of Variance (Anova). Obviously, we cannot show the sequence of mouse-clicks that did the



```
=> Anova - Two Way (Y(s)) ERR-RATE (X 1(s)) ETA-VAR (X 2(s)) PRED-PT
   Source     Degrees of Freedom   Sum of Squares   Mean Square   F Ratio      P
Interaction                    4            .0768         .0192   10.0912  .0000
 ETA-VAR                       2            .0061         .0031    1.6032  .2032
 PRED-PT                       2            .5517         .2759  144.9000  .0000
    Error                    261            .4969         .0019
    Total                    269           1.1316

Cell means   .0500   .1500   .2500
         2   .1565   .1259   .1288    .4113
         4   .2609   .2355   .2369    .7333
         6   .1842   .2148   .2439    .6429
             .6016   .5762   .6097   1.7875

Row-plot     GRAPH-DATA-ICON-1
```

Figure 4: Excerpt from sample interaction with CLASP

analysis, but Figure 4 shows the CLASP screen afterwards. The data show that there is a significant interaction between the two factors ($F = 10.09, p = 0.0$), because increasing variance didn't affect the error rate much when predicting two and four days in advance, but greatly increased the error rate when predicting six days in advance. We have superimposed a CLASP-generated graph to depict the interaction; note that one of the lines slopes upward, while the other two decline slightly. The data also show that, overall, the point of prediction was highly significant ($F = 144.9, p = 0.0$), but the amount of variance in the ship speed was not ($F = 1.6, p = 0.2$).

4.6 Current status

CLIP and CLASP may be obtained by anonymous ftp from ftp.cs.umass.edu. CLIP can be found under the directory pub/eksl/clip; CLASP is under pub/eksl/clasp. Manuals are included in both of these directories. A tutorial on CLASP is available under pub/eksl/clasp-tutorial.

Development of CLIP/CLASP continues, and is largely driven by user demand. We will continue to add useful statistical tests and data manipulation functions. Known limitations include problems with encapsulated POSTSCRIPT output from CLASP due to CLIM 1.1 and cosmetic glitches in display and input editing, which are due in part to CLIM 1.1. CLIP could use a graphical user interface for defining experiments. Comments, bugs and new feature requests can be sent to clasp-support@cs.umass.edu.

4.7 Conclusion

The purpose of this paper is to demonstrate how CLIP and CLASP can help in doing experimental studies in Artificial Intelligence generally, using an example grounded in transportation planning. CLIP works directly with a user's simulator, helping the experimenter define the dependent measures, control the independent variables and run the experiment. CLASP is a statistics package and as such competes with many good statistics packages on the market. Its advantages are that it is implemented in Common Lisp and CLIM, so that it can easily be combined with your simulator and with CLIP, allowing for a completely integrated experimental environment. We believe that such support for empirical science will be of significant benefit to the AI community.

Acknowledgments

This research is supported by ARPA/Rome Laboratory under contracts #F30602-91-C-0076 and #F30602-93-C-0100. The US Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. We thank Tim Oates for the use of his code and data, and Rob St. Amant for a helpful reading of the paper.

References

[1] Scott D. Anderson, Adam Carlson, David L. Westbrook, David M. Hart, and Paul R. Cohen. Clasp/Clip: Common Lisp Analytical Statistics Package/Common Lisp Instrumentation Package. Technical Report 93-55, University of Massachusetts at Amherst, Computer Science Department, 1993.

[2] Bolt Beranek and Newman, Inc. and ISX Corporation. Common prototyping environment testbed release 1.0: User's guide, 1993. BBN Systems and Technologies, 10 Moulton Street, Cambridge, MA 02138.

[3] Bradley Efron and Gail Gong. A leisurely look at the bootstrap, the jackknife, and cross-validation. *The American Statistician*, 37(1):36–48, February 1983.

[4] Bradley Efron and Robert Tibshirani. Statistical data analysis in the computer age. *Science*, 253:390–395, July 1991.

[5] Tim Oates and Paul R. Cohen. Humans plus agents maintain schedules better than either alone. Technical Report 94-03, University of Massachusetts at Amherst, Computer Science Department, 1994.

[6] Tim Oates and Paul R. Cohen. Mixed-initiative schedule maintenance: A first step toward plan steering. In Mark H. Burstein, editor, *ARPA/Rome Laboratory Knowledge-based Planning and Scheduling Initiative Workshop Proceedings*. Advanced Research Projects Agency and Rome Laboratory, February 1994. Also available as Technical Report 94-31, University of Massachusetts Computer Science Department.

[7] Tim Oates and Paul R. Cohen. Toward a plan steering agent: Experiments with schedule maintenance. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, 1994. Also available as Technical Report 94-02, University of Massachusetts Computer Science Department.

5 Monitoring in Embedded Agents

Marc S. Atkin and Paul R. Cohen

Abstract

Finding good monitoring strategies is an important process in the design of any embedded agent. We describe the nature of the monitoring problem, point out what makes it difficult, and show that while periodic monitoring strategies are often the easiest to derive, they are not always the most appropriate. We demonstrate mathematically and empirically that for a wide class of problems, the so-called "cupcake problems", there exists a simple strategy, *interval reduction*, that outperforms periodic monitoring. We also show how features of the environment may influence the choice of the optimal strategy. The paper concludes with some thoughts about a monitoring strategy taxonomy, and what its defining features might be.

5.1 Introduction

Monitoring is ubiquitous. Although often taken for granted, monitoring is in fact a necessary task for any agent, be it a human, a bumblebee, or an AI planner. Every embedded agent must query its environment. If we are to understand how to design agents and discover general laws of agent behavior, we must understand the nature of monitoring. This paper represents a summary of the work we have done on discovering and categorizing monitoring strategies. We want to demonstrate that the prevalent scheme, monitoring after each effector action [6, 7, 8, 9, 15], is not always the most efficient. The emphasis will be on discussing what makes certain monitoring problems more difficult than others, and investigating the tradeoff between two monitoring strategies, periodic and interval reduction.

In robotic systems, monitoring happens on at least three levels. On the lowest level, physical sensors probe the environment. The information they provide will generally have to be processed before it is usable to the agent. Higher up, some sort of reactive controller might be frequently checking the pre-processed sensor data to decide whether conditions exist to which the agent must immediately respond. Higher still, a symbolic plan that is controlling the robot might require confirmation that an action's precondition holds. This condition might be quite complicated in terms of the raw sensor data; for example: "is there another agent in my vicinity?" It might be the result of a lot of computation and maybe multiple pollings of the environment on the part of the lower level sensory sub-systems. Each of these levels requires a monitoring strategy, but due to the different circumstances and costs involved, the strategies might be completely different. A distinction is frequently made between *sensing* and *monitoring* (e.g. [8, 9, 15]). Sensing refers to the typically low-level data acquisition mechanisms needed to keep a world model up-to-date; monitoring involves querying this world model.

The multi-level model of monitoring also explains why interrupts cannot be used to circumvent the need for monitoring. A monitoring problem cannot be avoided; all you can do is

Submitted to *Fourteenth International Joint Conference on Artificial Intelligence* (IJCAI-95).

delegate it to a separate sensory system and instruct it to interrupt you when the event being monitored for occurs. The question of how this other sensory system should efficiently monitor for the event still has to be answered. And of course, introducing a new system for every possible event incurs a significant cost, be it in hardware for dedicated processors or execution time for interleaving several monitoring processes. There is also the problem of informedness. The process that monitors explicitly has potentially more information at its disposal—and can therefore do a better job at minimizing costs—than an interrupting process that only knows the interrupt criteria it was given when it was initiated and cannot take changing situations into account.

As we will use the term, a *monitoring strategy* is the scheme by which monitoring actions are scheduled. The strategy must balance the cost of monitoring against the cost of having inaccurate information about the environment. While in principle nothing more than an optimization problem, it is by no means trivial. Three factors make the problem difficult: First, the time between monitoring actions will depend on many factors, features of the environment and costs, some of which are not known or only known probabilistically. Second, these factors can be dynamic, changing over the course of a trial. Third, and perhaps most importantly, monitoring actions are not necessarily independent. The optimal placement of a monitoring action cannot always be computed without knowing the placement of all the successive ones.

The rest of the paper will be structured as follows: We will show that optimal periodic monitoring strategies are easy to derive; however, they are not always optimal. Better strategies, such as *interval reduction*, do not lend themselves to such a simple analysis. We will then go on to describe how we discovered monitoring strategies with a genetic algorithm. This algorithm was also used to compare periodic and interval reduction in a certain class of scenarios. This comparison is augmented by the sketch of a proof that shows the asymptotic superiority of interval reduction. The paper concludes with some thoughts about the features that should be included in a taxonomy of monitoring strategies.

5.2 Periodic Monitoring

The simplest and most common monitoring strategy is periodic monitoring: monitor every r time units. It is easy to show that if you have no knowledge about how the event being monitored for is temporally distributed, and if monitoring tells you only whether the event has occurred or not, then periodic monitoring is the best strategy [3]. Periodic monitoring is also the best strategy if the event can occur at any given time with equal probability [3]. As an example, consider fires breaking out in a forest. Let the probability that a fire breaks out on any given day be p; a fire incurs a cost F for every day it burns undetected. Assume that checking for fires requires sending a helicopter out to fly over the forest, and is therefore very costly. Call this cost H. What monitoring period r will minimize the combined expected cost of fire burning and monitoring? More accurately, what r will minimize the expected combined cost *per unit time*, since a trial can go on for an indefinite amount of time?

If one waits r days before monitoring the first time, the combined cost per unit time interval is

$$\begin{aligned} & \frac{1}{r}(H + F\sum_{i=1}^{r} p(r-i)) \\ = \; & \frac{1}{r}(H + Fp\frac{r^2+r}{2}) \qquad (1) \end{aligned}$$

Optimizing for r, one gets the period $\sqrt{2H/Fp}$. Since the monitoring epochs (an epoch is one instance of sending out the helicopter) are independent, this is the optimal period not just for the first epoch, but for the whole trial.

The problem becomes slightly more complicated if the probability p of fire breaking out varies over time. If this distribution is known to the agent, it can still compute a good monitoring strategy, but it will no longer generally be periodic.[6] But consider the case where the agent knows only one fire will occur in a finite amount of time. When the agent monitors, the conditional distribution of the fire changes, allowing the agent to acquire *more* information about when the fire is liable to start by monitoring. If the fire has not yet started, then the agent knows the fire must occur in the future; if it has, the agent can stop monitoring. This type of problem is called a *sequential decision problem.* It is impossible to schedule one monitoring event optimally without taking the others into account. We believe this is a general rule: If monitoring changes the agent's knowledge about the distribution of an event, monitoring events are no longer independent and monitoring becomes a sequential decision problem.

5.3 Interval Reduction

In 1985, Ceci and Bronfenbrenner conducted an experiment with children from two different age groups [4]. They instructed the children to take cupcakes out of an oven in 30 minutes, and to spend the time until then playing a video game. Behind the child was a wall clock, which the child could check, but checking it (the act of monitoring) was a distraction from the game, and therefore had an associated cost. In the first few minutes, all children checked the clock frequently. Ceci and Bronfenbrenner interpreted this as a "calibration" phase for the children's internal clock. Later, however, the ten-year-olds monitored approximately periodically, whereas the fourteen-year-olds monitored more frequently as the deadline approached.

We have termed this second strategy *interval reduction*, since the time to monitor next is reduced after each monitoring event. We use "cupcake problem" to denote any scenario in which an agent must monitor for a deadline and accumulates an error while making progress towards this deadline. In the previous example, the error accumulated due to a human's inaccurate internal clock. However, the same strategy arises if one assumes the internal clock

[6] The algorithm for this is quite straightforward: set up the function expressing combined cost of monitoring once plus the expected cost for not monitoring per unit time, starting at the current time. Monitor again at the local minimum of this function. This minimum can be determined numerically.

is accurate, but the information given to the agent is not. Like periodic monitoring, interval reduction is a very general strategy. We have shown empirically that it arises in a variety of agents, artificial and natural, in cupcake problems [5].

Interval reduction emerges if one simply tries to fix the probability of overshooting the goal during each epoch. Let's assume when the agent plans to wait t time steps, it actually waits $w(t)$, where w is a random variable normally distributed around t with standard deviation $\sigma = \sqrt{tm}$; m is a parameter describing how inaccurate the internal clock is.[7] The probability that the agent will wait more than

$$w(t)_\alpha = t + z_{2\alpha}\sigma$$

time units is exactly α, where $z_{2\alpha}$ is the number of standard deviations above the mean of a normal distribution that cuts off the highest $100\alpha\%$ of the distribution. To ensure that the agent does not exceed the desired deadline, D, with greater than α probability, we set $w(t)_\alpha = D$ and solve the previous equation for t:

$$t = D + \frac{m^2 z_{2\alpha}^2}{2} - m z_{2\alpha}\sqrt{D + \frac{m^2 z_{2\alpha}^2}{4}} \tag{2}$$

This equation tells us to wait a certain variable fraction of the current distance remaining to the deadline, D, after each monitoring event. This is an interval reduction strategy.

This approximate solution to the cupcake problem does not take the cost of monitoring or overshooting the deadline into account. If one does, the problem of finding the optimal solution turns out to be very difficult, because one cannot optimally place the next monitoring action without knowing whether or not there will be another chance to monitor again after that. If so, one might be more cautious this time around, for fear of overshooting the deadline; if not, one might take a larger step now. Again, we have a sequential decision problem. Eric Hansen has generated the optimal strategy for a given cupcake problem using a dynamic programming algorithm, and it is in fact an interval reduction strategy [11]. One computes the optimal placement of the last monitoring event and works backwards to the starting point. This algorithm has been extended to the general problem of monitoring plan execution [12].

The problem with a dynamic programming approach is that it gives you the monitoring strategy implicitly: at each time step, you know how long you should wait before monitoring again. You do not have a procedural representation of the strategy, such as given by equation 2, or even the general injunction to monitor periodically. Deriving the optimal strategy mathematically is not always an option due to the complexity of some of the optimization problems. We were therefore looking for an automatic, machine learning approach of generating *explicit* monitoring strategies for arbitrary problems. A genetic programming algorithm fit this billing.

[7]One obtains this normal distribution if one assumes that on each time step of the agent's internal clock, the agent actually waits $1 \pm m$ time steps, each with equal probability.

5.4 Periodic vs. Proportional Reduction

Motivated by the Ceci and Bronfenbrenner study, one of our main interests was the relative merits of interval reduction and periodic monitoring in the cupcake problem. In this section, we will present both empirical and analytical work (see [1] for a more complete description). The sub-class of interval reduction strategies under investigation is perhaps the most simple, characterized by a linear interval reduction function: after monitoring, the agent waits a fixed proportion of the distance remaining to the deadline before monitoring again. We call this class of strategies *proportional reduction* strategies.

5.4.1 The Genetic Programming Algorithm

Genetic programming concerns itself with the evolution of *programs* via a genetic algorithm (e.g. [13, 14]; see [10] for an introduction to genetic algorithms). In our case, these programs represent monitoring strategies. But they are more than just that: they also tell the agent how to solve the task at hand. They are representations of *behavior*.

Our goal was to apply genetic programming to a wide variety of problems, and build a taxonomy from the resulting monitoring strategies based on general features of the task and environment [3]. For this purpose, we needed a programming language that was general and easily extendable. We did not want to limit ourselves to monitoring problems in the temporal domain, and therefore chose as our testbed a two-dimensional world with obstacles and other types of terrain, within which a simple simulated mobile robot, equipped with a few sensors, must operate. There were two types of constructs in our behavior language: those that actually trigger an effector action, and those that steer the control flow within the program. Examples for the former category are the MOVE command, which moves the robot forward by a small distance in the direction it's currently facing, or the MONITOR command, which makes the robot poll a specified sensor. Examples for the latter category are LOOP commands and conditionals.

We developed several systems that learned monitoring strategies (see [2] for an overview). The one we discuss here, linear LTB, represented individuals as fixed-size lists of commands. Population size was 800, tournament selection with a tournament size of two was the selection scheme. A program's fitness score was based on the average of twelve different test cases. To account for the random and potentially sub-optimal performance of the genetic algorithm, we ran LTB ten times in each scenario. The best output of all these runs is taken to be the final output of the system for a given scenario.

To construct an instance of the cupcake problem in LTB's spatial world, we gave the robot the task of getting as close to a particular position in the map as possible, the *goal point*, without hitting it. The fitness function reflected this: it gave a penalty that was a quadratic function of the distance between the agent's final position on the map and the goal point. However, actually stepping onto the goal field was penalized highly, with a large constant fitness penalty (this was implemented by placing an obstacle on the goal field). Since overshooting the goal is treated differently than undershooting it, this problem is in fact an *asymmetric* cupcake problem.

```
Main program:
 NOP
 NOP
 TURNLEFT
 NOP
 LOOP 4 time(s):
   LOOP 1 time(s):
     TURNTOGOAL
     NOP
     NOP
     NOP
     MONITOR: object_distance
     NOP
     DISABLE: reached_goal
     NOP
     NOP
     MOVE
     IF (object_distance) <= .5 THEN STOP
     NOP
     LOOP (object_distance)+1 times:
       LOOP 4 time(s):
         NOP
         MOVE
*reached_goal* interrupt handler:
*hit_object* interrupt handler:
 IF (object_distance) <= 4.6 THEN STOP
 TURNLEFT
 NOP
 NOP
*object_distance* interrupt handler:
 DISABLE: hit_object
 MOVE
 NOP
 WAIT
```

Figure 5: A proportional reduction strategy generated by linear LTB

The robot had an accurate movement effector (movement corresponds to waiting in the original cupcake problem), but an inaccurate distance sensor. The error was modeled by adding a normal noise term to the value returned by the "object_distance" sensor, the "sonar" that gives the agent the distance to an obstacle in front of it. Since there was an obstacle on the goal field (and nowhere else), this sensor now returns the distance to the goal. The variance of the noise was proportional to the distance remaining to the goal point (this models the fact that errors in the sensor will accumulate with the size of the distance measured) and a noise parameter, m, that could be varied to change the degree of inaccuracy in the sensor. In this scenario, the robot has two other sensors, both of which are automatically updated: reached_goal ("have I reached the goal field?") and hit_object ("have I hit an obstacle?").

An illustrative monitoring strategy evolved by LTB is given in figure 5. The top half of the code is the main body of the program; the bottom half consists of the three *interrupt handlers* corresponding to the agent's sensors; however, in this scenario, they are not used.[8] Within the program's outer loop, "LOOP 4 times", the "MONITOR: object_distance" command is executed repeatedly. Nested within this loop is a second one, "LOOP (object_distance)+1 times", which takes the current distance to the goal (stored in the variable "object_distance" after monitoring) and loops over its value. Four MOVE'S are executed in this loop, each moving .1 distance units. Therefore, the proportionality constant is .4: the robot will move .4 times its estimate of the distance remaining before monitoring again. The command "IF (object_distance) $\leq$ .5 THEN STOP" terminates the program when the obstacle has reached a critical distance, and the TURNTOGOAL points the robot towards the goal. Note that this program is not pure proportional reduction, because there is one MOVE statement in the main loop that gets executed independently of the distance monitored.

5.4.2 Genetic Algorithm Results

We suspected that several features of the environment and the robot would influence the monitoring strategy, and it was the goal of the experiment described here to determine what that influence was. We had the following hypotheses:

1. If the monitoring error, m, is 0, a robot should monitor once only.
2. As m increases, the agent should be forced to take more cautious steps, i.e. the proportionality constant prc should decrease. The proportionality constant is the proportion of the robot's estimated distance to the goal that the robot actually moves before monitoring again.
3. Periodic monitoring becomes more likely as m increases (sonar data contains increasingly less useful information) and path length decreases (the advantage of proportional reduction, being able to traverse a distance with only a logarithmic number of monitors, is reduced).

We ran the genetic algorithm on five levels of monitoring error m (0, 0.5, 1.0, 2.0, 3.0) and three ranges of path length (1-4, 5-10, and 12-18 fields). The path length had to be a range or the genetic algorithm will simply adapt to move exactly the required distance, without monitoring at all. The best overall program for a scenario was re-tested on 36 start-goal pairs in each specified length range. Four measures were averaged over the 36 trials to evaluate the program's performance and strategy: fitness, total cost of monitoring, percentage of distance traversed while using proportional reduction, and the proportional reduction constant prc. Note that the second measure basically tells us how many times the agent monitored, as monitoring has a fixed cost.

Figure 6 shows the average fitness values for the final program on the 36 test pairs. For the most part, fitness values decrease monotonically as m and path length increase, with

[8] Interrupt handlers are the mechanism by which a robot can react directly to external events. Each program contained an interrupt handler for each kind of sensor the robot had. They executed automatically when the corresponding sensor value changed.

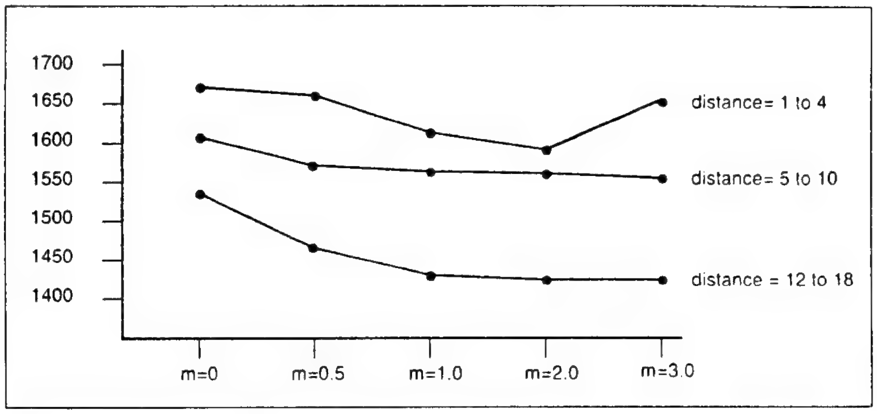


Figure 6: A plot of cell means for a two-way ANOVA, monitoring error m by distance. The dependent measure is the *fitness* of the best individual.

length	$m = 0.0$	$m = 0.5$	$m = 1.0$	$m = 2.0$	$m = 3.0$
1-4	PR^2 (**)	PR (.7)	PR^2 (*) (**)	PR (.5) (**)	check (*) (**)
5-10	PR^2	DPR	PR (.4)	- (*)	- (*)
12-18	DPR	PR (.7)	DPR (*) (**)	DPR	DPR

Key:		
	PR (c):	proportional reduction strategy with proportional reduction constant
	PR^2:	moves a proportion of the squared distance remaining
	DPR:	disproportionate reduction: moves a proportion of distance remaining plus a constant distance.
	check:	stops if obstacle is visible after monitoring
	–:	no monitoring strategy
	(*)	deliberately moves past goal to one side
	(**)	strategy is only capable of monitoring once

Table 10: The best individual's monitoring strategy

the exception of the $m = 3.0$, path length=1-4 situation. Both these effects were predicted: Longer path lengths mean higher energy consumption, which lowers fitness, and higher values of m mean more elaborate and energy-consuming monitoring strategies. Note however that path length has a much greater effect than monitoring error. Apparently the genetic algorithm copes quite well with finding good monitoring strategies. An ANOVA shows a clear effect of path length and monitoring error on fitness (for both main effects $p < 0.0001$), but no interaction effect, because both factors are independent variables.

We had expected to see monitoring cost increase as path length and especially monitoring error increases. This prediction is only partly verified. The expected pattern is distorted because when the monitoring error gets too high, the genetic algorithm does not come up with a monitoring strategy at all, but instead tries to find a different solution such as deliberately moving past the goal point (to the left or right) for a distance that is approximately the average of the path lengths of its training set. Table 10 gives an overview of the best individual's monitoring strategy for each test case. As one can see, in cases $m = 3.0$, path length 5-10; and $m = 2.0$, path length 5-10, no monitoring strategy was found. Sometimes (case $m = 1.0$, path length 1-4 and path length 12-18), it will combine the strategy of moving past the obstacle with proportional reduction, ensuring that if proportional reduction fails, it will still not touch it.

Interestingly, no program monitors more than once when the path length is very small. We had expected the "monitor only once" strategy only in the absence of sensor noise, but apparently, over such short distances, the extra effort involved in coming extremely close to the goal is not worth the energy necessary to achieve it. Another surprise is that for longer path lengths and no sensor error, programs still monitor more than once. It is hard to imagine how this could be advantageous.

An interval reduction strategy was found in nearly all situations. Usually this was not pure proportional reduction, but a slight variant (often an agent would move a small fixed distance within the monitoring loop or prior to it). Our hypothesis that periodic monitoring would become dominant for high values of monitoring error was not confirmed. It is important to mention, however, that periodic monitoring was indeed discovered by the algorithm in several high-monitoring-error situations, but its fitness was slightly lower than that of proportional reduction. The one clear exception is the $m = 3.0$, range = 1 - 4 case. Here, the robot moves a certain distance, then monitors for the obstacle. If the obstacle is visible ahead, it stops; otherwise it moves a fixed distance further, curving around the obstacle. This is notable because it is not a proportional reduction strategy, the sensor data are too inaccurate; the sensor is used only to detect the presence or absence of the obstacle.

Our second hypothesis was that the proportionality constant *prc* should decrease as m grows. Of course this constant is only really defined in cases of unmodified proportional reduction, which limits our available data points. But the few applicable cases do show the predicted effect (see Table 10). The range of the constant decreases from .7 in two of the $m = 0.5$ cases to .4 in the second $m = 1.0$ case. The constant of .5 in the $m = 2.0$, path length = 1 - 4 case should not be weighed too strongly as the robot only monitors once here. Even so, it is still close to the previous .4 value.

The hypothesis that periodic monitoring will surpass proportional reduction as m and path length increase appears to be refuted. Periodic monitoring never beat proportional reduction, although it did come close in the $m = 2.0$ cases. We are led to believe that periodic monitoring will only be advantageous when virtually no distance information is given. This is the main result of this experiment: some form of interval reduction seems to be appropriate for nearly all instances of the cupcake problem.

5.4.3 A Proof of Interval Reduction's Superiority

While it is certainly intuitive that proportional reduction should outperform periodic monitoring for the cupcake problem, a rigorous mathematical proof eluded us for a long time. A cupcake problem is characterized by three parameters, starting distance D, an error m, and monitoring cost *mon_cost*, and it is difficult to show that for *all* values of these parameters proportional reduction is superior, since the best strategy is a function of these variables. In fact, we assumed that for certain extreme values, such as short starting distances or large m, periodic monitoring would do better.

In this section, we will outline a proof of proportional reduction's *asymptotic* superiority over periodic monitoring in the cupcake problem. We have been able to show that for any cupcake problem with large enough D, a proportional reduction strategy exists that does better than the best periodic one. The proof is fairly involved, and cannot be presented in its entirety here (see [3]). Basically, the proof proceeds as follows: Derive a lower bound on the expected cost of the best periodic monitoring strategy for a cupcake problem. A periodic monitoring strategy is characterized solely by the parameter *period*, and involves monitoring every *period* time or distance units until the sensor reports that one is within *period/2* units of the deadline. After this cost bound has been established, derive an *upper* bound on the expected cost for the best proportional reduction strategy. We are trying to show that a proportional reduction strategy exists that will do better than the best periodic one. By establishing an upper bound on the cost of the former, we are only making the problem harder for ourselves. This proportional reduction strategy is dependent on two factors, the proportional reduction constant *prc*, and a threshold *thresh*. One terminates the trial once the sensor reports that one is within *thresh* time or distance units of the deadline.

It should be pointed out that the proof is based on the variant of the cupcake problem that assumes the agent's sensor is totally accurate, but its movement (or waiting) effector is not. This effector has a normal error with standard deviation $\sigma = m\sqrt{t}$, where t is the distance (or time) the agent planned to move/wait. Also, the quadratic distance penalty is symmetric, i.e., the same cost is charged for overshooting and undershooting the deadline.

For both strategies, the total expected cost is computed by estimating the number of times the strategy will monitor, z, and adding to $z\cdot$ *mon_cost* the agent's squared expected distance from the goal when the trial ends. In the periodic case, the lower bound on the total cost turns out to be a relatively simple function,

$$\text{cost}_{\text{periodic}} \;\geq\; \frac{D}{\text{period}}\text{mon\_cost} + \text{period}\cdot m^2 \qquad (3)$$

Minimizing for *period*, we get

$$\text{cost}_{\text{periodic}} \;\geq\; 2m\sqrt{D}\sqrt{\text{mon\_cost}} \qquad (4)$$

The upper bound on the cost for proportional reduction is a lot more complicated. It is computed by finding the bound on the cost for each epoch (monitor and move sequence), and summing over the number of times monitored. We get

$$\text{cost}_{\text{pr}} \leq \log_{\frac{1}{1-\text{prc}}}\frac{D}{\text{thresh}}(\text{mon\_cost} + 22\cdot m^4\frac{\text{prc}^2}{(1-\text{prc})^2}) + \text{thresh}^2 \qquad (5)$$

In sum, the periodic cost is $\Omega(D^{\frac{1}{2}})$ and the proportional reduction cost is $O(\log(D))$. Proportional reduction will therefore beat periodic for a large enough D. The exact D value at which this occurs depends on m and *mon_cost*, and is typically in the range between 200 to 1000 distance units. For lower starting values, this proof does not show that proportional reduction is better. This does not mean, however, that the converse is actually true, and with tighter bounds on the costs, the minimal starting distance for which the proof holds can probably be reduced substantially.

5.5 A Taxonomy of Monitoring Scenarios and Strategies

We believe three factors determine an agent's behavior: the task it is given, the environment it is operating in, and the way it is built (its architecture). To understand monitoring, it will be necessary to understand the influence of these three factors. The experiment in section 5.4.2 was a small step in this direction: it analyzed the effects of two environmental factors, sensing error and starting distance, on the cupcake problem.

On a more abstract level, though, what will be the features upon which a monitoring strategy taxonomy is built? Here are a few candidates regarding task and environment: What shape is the penalty function? Is it symmetric, does it change over time? What is the type of task, what are the associated costs? Do several goals have to be achieved in parallel? Are they dependent on each other? Does monitoring for one goal provide information for another? Are monitoring acts independent? Is deciding when to stop monitoring part of the problem, or are trials of indefinite extent? How noisy and unpredictable is the environment? Is it a spatial or temporal domain? Does it allow actions to be reversed? (In the temporal cupcake problem, one cannot increase the time to the deadline, in the spatial version one can increase the distance by going backwards or turning around.) Do false actions have fatal consequences? Are other agents present?

With respect to the agent's architecture, we have features like: What types of sensors are available? Can they be used independently? What are their costs, their error functions? Is the error constant or a function of other environmental parameters? Can the effectors influence what is being sensed, or are they independent? What degree of control does the agent have over its environment? What is the agent's reaction time to external events? What is its computing power? Any of the above factors may influence the decision on what is the appropriate strategy. The hard problem is determining what subset of factors are important in a given scenario. We suspect that in the cupcake problem, only a few factors influence the choice of best monitoring strategy, the most important one being the amount of information returned by the sensor. Most other factors we looked at, including things like the shape of the penalty function or the monitoring cost, do not change the superiority of interval reduction. During our experiments with different scenarios, we have come to believe that there may only be a relatively small number of basic monitoring strategies [3]. This would imply that the factor subset cannot be very large.

An interesting way of looking at the difference between periodic and interval reduction is in terms of the amount of information the sensor provides. If the agent in the cupcake problem had no information other than whether it had overshot the deadline yet or not, it would be forced to use periodic monitoring. Based on this idea, we propose three *complexity classes* for categorizing monitoring problems. "Complexity" is not formally defined, but intuitively it is a combined measure of how much effort is required to solve these problems optimally.

1. *precomputable*: These are problems where the complete sequence of monitoring can be planned in advance. An example is monitoring for forest fires that occur with equal probability at any time. If the distribution of the event is known to the agent, it can precompute the optimal monitoring strategy.

2. *dynamically dependent on environment*: Some strategies will depend on the current state of the environment. An example of this is monitoring for fires under different weather conditions: Monitoring will always be periodic, but the period is dependent on the current weather, as this influences the chances of a fire starting. Because of this variable, the sequence of monitoring actions cannot be precomputed. It might often be possible, however, to express the optimal period in closed form, with the weather being simply one of the variables in the equation.

3. *sequential decision problems*: These are the most difficult problems. Not only is the monitoring rate dependent on the environment, but also on all future monitoring actions. An example for this is of course the cupcake problem. When to monitor again depends on the information returned by the sonar sensor, and also on the decision of whether or not to monitor *again* after the next movement cycle.

5.6 Summary

Monitoring is an important problem in agent design, one that is often overlooked. This paper has attempted to show that periodic monitoring is only the most simple of a whole range of different monitoring strategies. For many situations, in particular when the sensor returns more than just binary information, periodic monitoring is not necessarily the best strategy to use.

For virtually all instances of the cupcake problem, interval reduction appears to be superior to periodic. Using this example, we have shown how the environment can effect the strategy. We see this as the first step in constructing a monitoring strategy taxonomy. Many other factors will determine the choice of monitoring strategy, and we have listed some of the candidates. Determining which factors are relevant in a given scenario is still very much an open research question that we hope will be expanded on.

Acknowledgments

This work is supported by ARPA/Rome Laboratory under contract #'s F30602-91-C-0076 and F30602-93-C-0010. The U.S. government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

References

[1] Atkin, M. & Cohen, P. R., 1993. Genetic Programming to Learn an Agent's Monitoring Strategy, *Proceedings of the AAAI 93 Workshop on Learning Action Models*, pp. 36-41.

[2] Atkin, M. S. & Cohen, P. R., 1994. Learning Monitoring Strategies: A Difficult Genetic Programming Application. *Proceedings on the First IEEE Conference on Evolutionary Computation*, IEEE Press, pp. 328-332a.

[3] Atkin, M. S., 1994. A Genetic Programming Approach to a Monitoring Strategy Taxonomy. Master's Thesis, University of Massachusetts at Amherst.

[4] Ceci, S. J. & Bronfenbrenner, U., 1985. "Don't forget to take the cupcakes out of the oven": Prospective memory, strategic time-monitoring, and context. *Child Development*, Vol. 56. pp. 152-164.

[5] Cohen, P. R., Atkin, M. S., and Hansen, E. A., 1994. The Interval Reduction Strategy for Monitoring Cupcake problems, *Proceedings of the Third International Conference on the Simulation of Adaptive Behavior*. MIT Press, Cambridge, MA.

[6] Doyle, R., Atkinson, D., & Doshi, R., 1986. Generating Perception Requests and Expectations to Verify the Execution of Plans. *AAAI-86*, pp. 81-88.

[7] Fikes, R., Hart, P., & Nilsson, N., 1972. Learning and Executing Generalized Robot Plans. *Artificial Intelligence* **3** (4), pp. 251-288.

[8] Firby, R. J., 1987. An Investigation into Reactive Planning in Complex Domains, *AAAI-87*, pp. 202-206.

[9] Georgeff, M. P., & Lansky, A. L., 1987. Reactive Reasoning and Planning, *AAAI-87*, pp. 677-682.

[10] Goldberg, D. E., 1989. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley.

[11] Hansen, E. A., 1992. Note on monitoring cupcakes. *EKSL Memo #22.* Experimental Knowledge Systems Laboratory, Computer Science Dept., University of Massachusetts, Amherst.

[12] Hansen, E. A., 1994. Cost-Effective Sensing During Plan Execution. *AAAI-94*, pp. 1029-1035.

[13] Koza, J. R., 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection and Genetics.* MIT Press, Cambridge, MA.

[14] Koza, J. R. & Rice, J.P., 1992. Automatic Programming of Robots using Genetic Programming. *AAAI-92*, pp. 194-207.

[15] McDermott, D., 1978. Planning and Acting. *Cognitive Science* **2** (2), pp. 71-109.

6 Cost-Effective Sensing During Plan Execution

Eric A. Hansen

Abstract

Between sensing the world after every action (as in a reactive plan) and not sensing at all (as in an open-loop plan), lies a continuum of strategies for sensing during plan execution. If sensing incurs a cost (in time or resources), the most cost-effective strategy is likely to fall somewhere between these two extremes. Yet most work on plan execution assumes one or the other. In this paper, an efficient, anytime planner is described that controls the rate of sensing during plan execution. The sensing interval is determined by the state during plan execution, as well as by the cost of sensing, so that an agent can sense more often when necessary. The planner is based on a generalization of stochastic dynamic programming.

Introduction

The characteristic assumptions of classical planning — a complete and certain action model and deterministic plan execution — make sensing during plan execution unnecessary. A planner that knows the initial state of the world and can predict the effects of its actions with certainty has no reason for sensing. Hence, classical planners constructed open-loop plans.

The reactive approach pursued in recent planning research was developed for environments in which the assumptions made by classical planners are unjustified, in other words, for most realistic environments. In the real world, actions may not have their intended effects and the environment can change in unexpected ways. Because a planner cannot project the course of plan execution with certainty, the reactive approach is to construct a plan that specifies what action to take in each of many different states, and to sense the world after each action so that the agent can choose the next action based on the current state.

The question that is the starting point for this paper is whether it is cost-effective to sense the world after every action in a plan. There is often a cost, in time or resources, for acquiring and processing sensory information. If sensing is expensive, it may be better to sense less frequently, especially if there is not that much uncertainty about the state of the world during plan execution. Between sensing the world after every action (as in a reactive plan) and not sensing at all (as in an open-loop plan), lies a continuum of sensing strategies. The most cost-effective one is likely to fall somewhere between these two extremes.

Although the problem of sensing costs has been relatively neglected in planning research, two different approaches to it have been tentatively explored. In one, cost-effectiveness is achieved by sensing only a subset of the features of the environment (Chrisman & Simmons 1991; Tan 1991). This raises interesting issues, among them the problem of identifying a state from incomplete perception (Whitehead & Ballard 1991), but it still assumes an agent senses its environment after each action. The second approach allows an agent to sense at wider intervals than after every action, where the sensing interval is set based on factors such as the cost of sensing and the cost and likelihood of error. But in this approach, the simplifying assumption is usually made that the rate of sensing should be constant throughout plan execution. For example, Abramson (1993) gives a decision-theoretic analysis to show that a fixed sensing interval during plan execution is optimal, and a formula for calculating what the interval should be. Hendler and Kinny (1992) apply a similar analysis to TileWorld, and Langley, Iba, and Shrager (1994) also assume periodic sensing. All of these analyses are based on a model of plan execution in which errors occur, in Abramson's description, "spontaneously"; that is, they are no more likely in one state of the world than another, nor any more likely after one action than another. From the assumption that errors are spontaneous, it follows that a fixed rate of sensing is optimal. But in many realistic environments, some states are riskier than others and some actions more error-prone. As a result, the success of some parts of a plan will be less predictable than other parts, and this suggests that the frequency of sensing

In *Proceedings of the Twelfth National Conference on Artificial Intelligence* (AAAI-94). AAAI Press/MIT Press, pp. 1029-1035.

should change depending on which part of a plan is being executed.

When the likelihood of plan error depends on the current state or on the action taken in that state, sensing becomes a planning problem; that is, decisions about when to sense the environment must take into account the actions in the plan and the projected state during plan execution. In order to treat the problem of when to sense during plan execution as a planning problem, this paper adopts the framework of Markov decision theory. Besides providing mathematical rigor, this framework has proven useful for formalizing planning problems in stochastic domains (Dean et al 1993; Koenig 1992), as well as for relating planning to reinforcement learning (Sutton 1990; Barto, Bradtke, & Singh 1993). In a conventional Markov decision problem, a policy (or plan) is executed by automatically sensing the world after each action, without considering the cost this might incur. Because this is exactly the assumption being questioned in this paper, we begin by showing how to incorporate sensing costs and formulate sensing strategies in the framework of Markov decision theory. Then an efficient, anytime planner is described that can adjust the rate of sensing during plan execution, depending on the state and actions of a plan. The approach developed here is directly applicable to work on planning for stochastic domains using techniques based on dynamic programming, and is suggestive for work on integrating sensing with plan execution in general.

Including Sensing Costs in a Markov Decision Problem

A discrete-time, finite state and action Markov decision problem is described by the following elements. Let S be a finite set of states, and let A be a finite set of actions. A *state transition function* $P{:}S \times A \times S \rightarrow [0,1]$ specifies the outcomes of actions as discrete probability distributions over the state-space. In particular, $P_{xy}(a)$ gives the probability that action a taken in state x produces state y. A *payoff function* $R{:}S \times A \rightarrow \Re$ specifies rewards (or costs) to be maximized (or minimized) by a plan. In particular, $R(x,a)$ gives the expected single-step payoff for taking action a in state x. A *policy* $\pi{:}S \rightarrow A$ specifies an action to take in each state. The set of all possible policies is called the *policy space*. To weigh the merit of different policies, a *value function* $V_\pi{:}S \rightarrow \Re$ gives the expected cumulative value received for executing a policy, π, starting from each state. The value function for a policy satisfies the following system of simultaneous linear equations:

$$V_\pi(x) = R(x,\pi(x)) + \lambda \sum_{y \in S} P_{xy}(\pi(x)) V_\pi(y), \qquad (1)$$

where $0 \le \lambda < 1$ is a *discount factor* that gives higher importance to payoffs in the near future. A policy π^* is optimal if $V_{\pi^*}(x) \ge V_\pi(x)$ for all states x and policies π. *Dynamic programming* is an efficient way of searching the space of possible policies for an optimal policy, using the value function to guide the search. There are several versions of dynamic programming and the ideas in this paper can be adapted to any of them, but Howard's policy iteration algorithm is used as an example (Howard 1960). It consists of the following steps:

1. *Initialization*: Start with an arbitrary policy $\pi{:}S \rightarrow A$.
2. *Policy evaluation*: Compute the value function for π by solving the system of simultaneous linear equations given by equation (1).
3. *Policy improvement:* Compute a new policy π' from the value function computed in step 2 by finding, for each state x, the action a that maximizes expected value, or formally:

$$\forall x: \quad \pi'(x) = \arg\max_{a \in A} \left[R(x,a) + \lambda \sum_{y \in S} P_{xy}(a) V_\pi(y) \right].$$

4. *Convergence test*: If the new policy is the same as the old one, it is optimal and the algorithm stops. Otherwise, set $\pi := \pi'$ and go to step 2.

The policy evaluation step requires solving a system of simultaneous linear equations and so has complexity $O(n^3)$ using a conventional algorithm such as Gaussian elimination, and at best $O(n^{2.8})$, where n is the size of the state set. The complexity of the policy improvement step is $O(mn^2)$, where m is the size of the action set. The policy iteration algorithm is guaranteed to improve the policy each iteration and to converge to an optimal policy after a finite number of iterations. It is also an anytime algorithm that can be stopped before convergence to return a policy that is monotonically better than the initial policy as a function of computation time.

In the conventional theory of Markov decision problems described so far, each action takes a single time-step and the world is automatically sensed at each step to see what action to take next. We want to find some way to represent problems in which an arbitrary sequence of actions can be taken before sensing. To do so, we define "multi-step" versions of each of the elements of a Markov decision problem with the exception of the state space.

A *multi-step action set* $\bar{A}$ includes all possible sequences of actions up to some arbitrary limit z on their length, where the arbitrary limit keeps the action set finite. The bar over the A is used to indicate "multi-step." Note that the multi-step action set includes both primitive actions, which take a single time step, and sequences of primitive actions. A tuple $\langle a_1..a_k \rangle$ represents a k-length sequence of actions, where $1 \le k \le z$. A *multi-step transition function* $\bar{P}{:}S \times \bar{A} \times S \rightarrow [0,1]$ gives the state transition probabilities for the multi-step action set. The multi-step transition probabilities for an action sequence $\langle a_1..a_k\rangle$ can be computed by multiplying the matrices that contain the single-step transition probabilities for the actions in the sequence, or formally, $\overline{P}(\langle a_1..a_k\rangle) = P(a_1)...P(a_k)$, where the matrix $P(a)$ contains the single-step transition probabilities for action a. In particular, $\overline{P}_{xy}(\langle a_1..a_k\rangle)$ gives the probability that taking the sequence of actions $\langle a_1..a_k\rangle$ starting from state x results in state y. A *multi-step payoff function* gives the expected "immediate" payoff received in the course of a sequence of actions. It is the sum of the expected payoffs received each time step during execution of the action sequence, with a cost for sensing, C, incurred at the end (when the world is sensed again),[1] and is defined formally as follows:

$$\overline{R}(x,\langle a_1..a_k\rangle) = R(x,a_1) - \lambda^k C + \sum_{j=1}^{k} \lambda^j \sum_{y\in S} \overline{P}_{xy}(\langle a_1..a_j\rangle) R(y,a_j).$$

Note that the time step becomes the exponent of the discount factor to ensure discounting is done in a time-dependent way. The equation also shows that the cost of sensing can be controlled by varying the length of action sequences. A *multi-step policy* $\overline{\pi}:S \rightarrow \overline{A}$ maps each state to a sequence of actions to be taken before sensing again. The *multi-step value function* for $\overline{\pi}$ satisfies the following system of simultaneous linear equations,

$$\overline{V}_{\overline{\pi}}(x) = \overline{R}(x,\overline{\pi}(x)) + \lambda^{length(\overline{\pi}(x))} \sum_{y\in S} \overline{P}_{xy}(\overline{\pi}(x)) \overline{V}_{\overline{\pi}}(y), \quad (2)$$

where the length of an action sequence is again the exponent of the discount factor to ensure time-dependent discounting.

With multi-step versions of the action set, state transition function, payoff function, policy, and value function, we have a well-defined Markov decision problem for which policy iteration can find an optimal policy for interleaving acting and sensing. The multi-step version of the policy iteration algorithm is as follows:

1. *Initialization*: Start with an arbitrary policy $\overline{\pi}:S \rightarrow \overline{A}$.
2. *Policy evaluation:* Compute the value function for $\overline{\pi}$ by solving the system of simultaneous linear equations given by equation (2).
3. *Policy improvement:* Compute a new policy, $\overline{\pi}'$, from the value function computed in step 2 by finding, for each state x, the sequence of actions that maximizes expected value, or formally:

$$\forall x: \quad \overline{\pi}'(x) = \arg\max_{\langle a_1..a_k\rangle\in\overline{A}} \Big[\overline{R}(x,\langle a_1..a_k\rangle) + \lambda^k \sum_{y\in S} \overline{P}_{xy}(\langle a_1..a_k\rangle) \overline{V}_{\overline{\pi}}(y)\Big].$$

4. *Convergence test*: If the new policy is the same as the old one, it is optimal and the algorithm stops. Otherwise, set $\overline{\pi} := \overline{\pi}'$ and go to step 2.

The policy evaluation step still requires solving a system of n simultaneous linear equations and so has the same complexity as for a single-step Markov decision problem. However, the complexity of the policy improvement step has been dramatically increased. Naively performing the policy improvement step now requires evaluating, for each state, every possible sequence of actions up to the fixed limit z on the length of a sequence. Even assuming that all multi-step transition probabilities and payoffs are pre-computed,[2] the complexity of the policy improvement step would be $O(m^z n^2)$ where the factor m^z comes from the fact that there are approximately m^z different action sequences of length at least 1 and no greater than z, where m is the number of primitive actions. In other words, allowing a sequence of actions to be taken before sensing causes an exponential blowup in the size of the policy space and an apparently prohibitive increase in the complexity of the dynamic programming algorithm.

An Efficient Planning Algorithm

Although searching the space of all possible sequences of actions exhaustively for each state is prohibitive, a practical approach to the problem of cost-effective sensing is still possible within this formal framework. Many action sequences are implausible or counterproductive, either because actions lead away from the goal or because they do not make sense in a given state. By using the value function to estimate the relative merit of different sequences, the space of possible action sequences can be searched intelligently in order to find a good, if not optimal, multi-step policy.

The multi-step action set is first organized as a search tree in which each node corresponds to an action sequence, and the successors of a node are created by adding a single action to the end of the sequence. The root of the search tree corresponds to the null action (which we do not consider part of the action set), level 1 of the tree contains

[1]For simplicity, in this paper we assume that sensing has a fixed cost. It is a straightforward generalization to make the cost of sensing a function of state and/or action.

[2]The combinatorial explosion of the number of possible action sequences usually makes it prohibitive to precompute and store all multi-step transition probabilities and payoffs. Fortunately for the efficient algorithm described in the next section, the vast majority are never needed and those that are can be cached to avoid repeated recomputation.

all sequences of length 1, level 2 contains all sequences of length 2, and so on. An action sequence with a good expected value can be found efficiently by starting from the root and choosing a greedy path through the tree. At each node, the expected values of the successor nodes are computed and the path extended to the node with the highest expected value, as long as it is higher than the expected value of the current node. If no successor node is an improvement, the search is stopped. This corresponds to constructing an action sequence by starting with the null sequence and adding actions to the end of the sequence one at a time, using the following two heuristics:

greedy heuristic: Always extend a sequence of actions with the action that gives the resulting sequence the highest expected value.

stopping heuristic: Stop as soon as extending a sequence of actions with the best next action reduces the expected value of the resulting action sequence.

The stopping heuristic simply means to assume that if it does not pay to take one more action before sensing, it will not pay to take two or more actions before sensing either — a very reasonable heuristic. Note that the stopping heuristic makes it possible to stop the search before the arbitrary, fixed limit z on the length of an action sequence is reached.

In addition to these two search heuristics, the greedy search algorithm needs one more feature. A crucial property of policy iteration is that it guarantees monotonic improvement of the policy each iteration; in particular, it guarantees that the value of each state is equal to or better than the value for the same state on the previous iteration. This property, together with a finite policy space, guarantees that policy iteration converges in a finite number of iterations. However, because the efficient search algorithm searches the space of possible action sequences greedily, local maxima may lead it to an action sequence that has a lower value than the action sequence found on the previous iteration. To prevent the value of any state from decreasing from one iteration to the next, the following rule is invoked at the end of the greedy search:

monotonic improvement rule: If the value of the action sequence found by greedy search is not better than the value of the action sequence specified by the current policy, do not change the current action sequence.

This rule ensures that a state's value is always equal to or greater than its value on a previous iteration because if an improved action sequence is not found, the previous action sequence has a value at least as good.

The greedy search algorithm for the policy improvement step is summarized as follows:

For each state x:

1. Initialize $\langle sequence \rangle$ to the single action with the highest expected value.
2. Compute the expected value of each possible extension of $\langle sequence \rangle$ by a single action.
3. *Stopping heuristic*: If no extension has a higher expected value than the current sequence, stop the search and go to step 5.
4. *Greedy heuristic*: If at least one extension has an improved value, set $\langle sequence \rangle$ to the extension with the highest value and go to step 2.
5. *Monotonic improvement rule*: If the sequence found by greedy search has a higher expected value than the sequence specified by the current policy, change the policy for this state to $\langle sequence \rangle$, otherwise, leave the policy for this state unchanged.

From the monotonic improvement rule and the finiteness of the policy space, the multi-step version of policy iteration using the greedy search algorithm is guaranteed to converge after a finite number of iterations. However, it is impossible to say whether it will converge to a globally optimal policy or to one that is a local optimum. Nevertheless, the opportunity to vary the sensing interval during plan execution, based both on the state and the cost of sensing, makes it possible for the multi-step algorithm to converge to a policy that is better – often considerably better – than a policy that automatically senses each time step. Moveover, the greedy search algorithm can find a good multi-step policy efficiently, despite the exponential explosion of the policy space.

When the greedy search algorithm is used in the policy improvement step, the number of action sequences that must be evaluated for each state is reduced from m^z to km, where again, m is the number of different actions, z is the maximum length of an action sequence, and $k \leq z$ is the average length of an action sequence. In other words, the complexity of searching for an improved action sequence for a state is reduced from exponential to linear. If the multi-step transition probabilities and payoffs were already available, the average time complexity of the policy improvement step would be $O\left(kmn^2\right)$, only a constant factor greater than for the single-step algorithm. In most cases, however, the multi-step transition probabilities and payoffs are not given at the outset and must be computed from the single-step transition probabilities and payoffs. If they are computed on the fly each time they are needed, the time complexity of the policy improvement step is $O\left(kmn^3\right)$. This can be alleviated somewhat by saving or caching the multi-step transition probabilities and payoffs the first time they are computed. Because most regions of the search space are never explored, most multi-step transition probabilities and payoffs do not need to be computed. Those that do tend to be re-used from iteration to iteration, making caching very beneficial. With caching, the time complexity of the policy improvement step becomes $O\left(pkmn^2\right)+O\left((1-p)kmn^3\right)$, where p is the proportion of cache hits on a particular iteration. Because the proportion of cache hits tends to increase from one iteration to the next, the policy improvement step runs faster as the algorithm gets closer to convergence.

Generating the multi-step transition probabilities and payoffs is by far the most computationally burdensome aspect of the multi-step algorithm. Nevertheless, because the time complexity of the policy evaluation step is already $O\left(n^3\right)$, or at best $O\left(n^{2.8}\right)$, the overall asymptotic time complexity of a single iteration of the multi-step algorithm, consisting of both policy evaluation and improvement, is still about the same or only a little worse in order notation, although the constant is greater for the multi-step algorithm. This is important because it means that scaling up dynamic programming to problems with large state spaces, the characteristic weakness of dynamic programming, is not made appreciably more difficult by the multi-step algorithm.

Of course, it takes longer for the multi-step version of policy iteration to converge than for the single-step version. However, both are iterative (anytime) algorithms that can have a good policy ready before convergence. Moreover, it is easy to ensure that multi-step policy iteration always has a policy as good or better than the single-step algorithm in the same amount of computation time. Simply run the single-step algorithm until convergence, then continue from that point with the multi-step algorithm. This approach is used in the example described in the next section. In most cases, computing the optimal single-step policy before beginning the multi-step algorithm leads to faster initial improvement of plan quality because it postpones the expense of computing multi-step probabilities and payoffs. For much the same reason, gradually adjusting upward the maximum allowable length of an action sequence from one iteration of the multi-step algorithm to the next can accelerate convergence.

A Path-Planning Example

The multi-step policy iteration algorithm can be used for any planning problem that is formalizable as a Markov decision problem. The following path-planning example provides a very simple illustration. Imagine a robot in the simple grid world shown in figure 1. Each cell in the grid represents a state. The actions the robot can take to move about the grid are {North, South, East, West}, but their effects are unpredictable. Whenever the robot attempts to move in a particular direction, it has 0.8 probability of success. However, with 0.05 probability it moves in a direction that is 90 degrees off to one side of its intended direction, with 0.05 probability it moves in a direction that is 90 degrees off to the other side, and with 0.1 probability it does not move at all. For example, if it attempts to move north it is successful with 0.8 probability, but it moves east with 0.05 probability, moves west with 0.05 probability, and does not move at all with 0.1 probability. If the robot hits a wall, it stays in the same cell. In this problem, the robot cannot be sure where it has moved without sensing, and becomes less sure about where it is the more it moves without sensing. Sensing its current state has a cost of 1. Hitting a wall has a cost of 5. Therefore, this is a cost-minimization problem in which the robot must balance the cost of sensing against the value of knowing where it is as it tries to find its way to the goal. Costs stop accumulating and the problem ends when the robot senses that its current state is the goal state. The discount factor used is 0.99999.

Figure 2 shows the anytime improvement of plan quality with multi-step policy iteration, beginning from the point at which the conventional, single-step algorithm converges. The multi-step algorithm converges to a policy that performs better than the single-step policy by almost a factor of two. The numbers in the grid cells of figure 3 give the sensing interval for each state for the multi-step policy at convergence. Due to lack of space, only one of the action sequences is shown. The others do what one would expect: move the robot towards the goal. In the action sequence shown, the robot moves away from the wall first, to avoid accidentally hitting it, before moving through the middle of the room towards the goal.

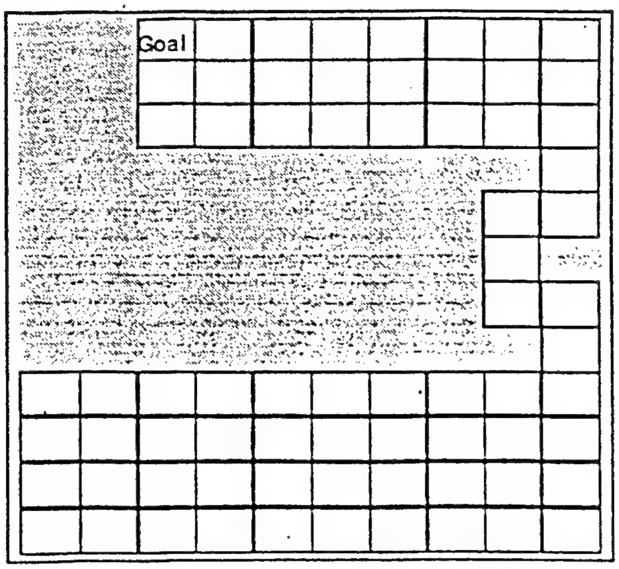


Figure 1: In this simple grid world, the robot tries to minimize the combined cost of sensing and hitting walls on its way to the goal.

Note that in the west side of the large room on the bottom of the grid world, the robot senses less frequently because there is less danger of hitting a wall, and then it senses more frequently as it approaches the east wall where it must turn into the corridor. The sharp corners of the narrow corridor cause the robot to sense nearly every time step. When it enters the room with the goal, it senses less frequently at first, given the wider space, then more frequently as it approaches the goal, to make sure it enters the goal state successfully. This sensing strategy is an intuitive one. A person would do much the same if he or she tried to get to the goal blindfolded, and was charged for stopping and removing the blindfold to look about.

Conclusion

This paper describes a generalization of Markov decision theory and dynamic programming in which sensing costs can be included in order to plan cost-effective strategies for sensing during plan execution. Within this framework, an efficient search algorithm has been developed to deal with the combinatorial explosion of the policy space that results from allowing arbitrary sequences of actions before sensing. The result is an anytime planner that can adjust the rate of sensing during plan execution in a cost-effective way.

There are some limitations to this approach. First, it applies specifically to planners that are based on dynamic programming techniques, although it is suggestive for the problem of interleaving sensing and acting in general. Second, it assumes a simplistic sensing model in which sensing is a single operation that acquires a perfect snapshot of the world. Many real-world problems involve sensor uncertainty and multiple sensors that each return a fraction of the information available. Nevertheless, the simple sensing model assumed here is often useful and is assumed by a number of planners and reactive controllers. In contexts in which this sensing model is appropriate, the anytime planner described here provides an efficient method for dealing with sensing costs.

Finally, the formal framework within which this approach is developed makes it possible to confirm a very natural intuition. The rate of sensing during plan execution should depend on the cost of sensing and on the projected state during plan execution. This makes it possible for an agent to sense less frequently in "safe" parts of a plan, and more frequently when necessary, as the path-planning example illustrates.

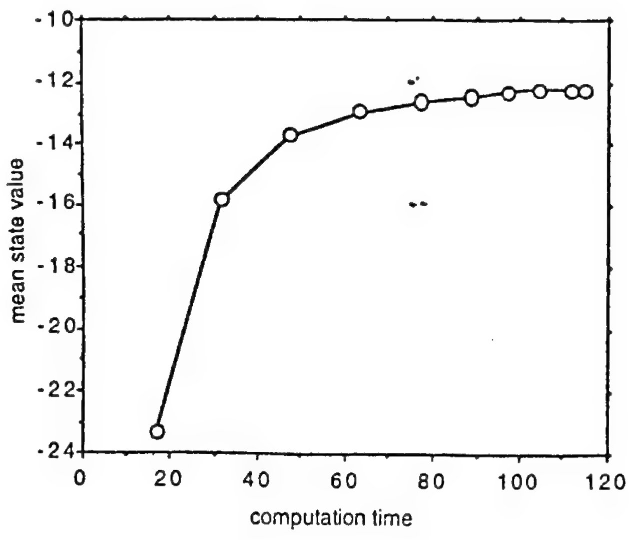


Figure 2: Anytime improvement in plan quality with multi-step policy iteration for the grid world example of figure 1. The single-step algorithm is started at time zero and the first circle marks the point at which it converges and the multi-step algorithm begins. Each circle after that represents a new iteration of the multi-step algorithm. The iterations are closer together near convergence because of previous caching of multi-step transition probabilities and payoffs. Computation time is in cpu seconds.

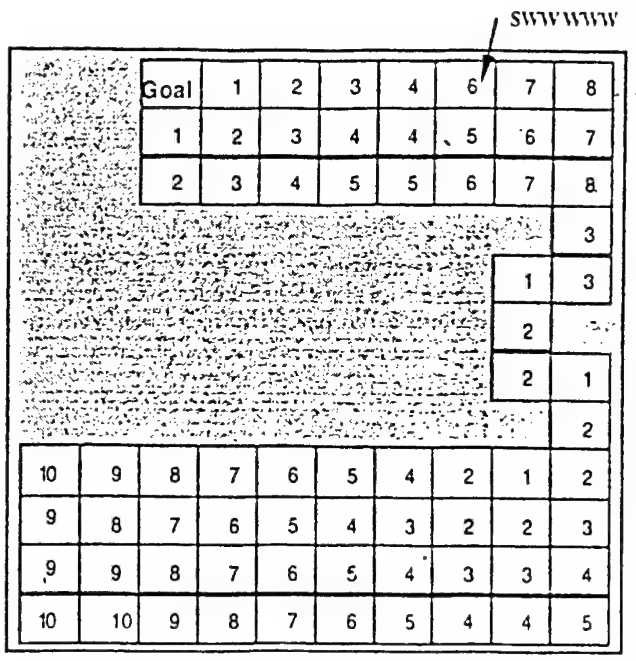


Figure 3: Multi-step policy at convergence. Each cell (or state) contains the length of the action sequence to take from that state before sensing again. One of the action sequences is shown as an example.

Acknowledgements

This research is supported by ARPA/Rome Laboratory under contract #F30602-91-C-0076 and under an Augmentation Award for Science and Engineering Research Training. The US Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

The author is grateful to Scott Anderson for valuable help and contributions to this work, to Paul Cohen for supporting this research, and to Andy Barto and the anonymous reviewers for useful comments.

References

Abramson, B. 1993. A Decision-Theoretic Framework for Integrating Sensors into AI Plans. *IEEE Transactions on Systems, Man, and Cybernetics* 23(2):366-373. (An earlier version appears under the title "An Analysis of Error Recovery and Sensory Integration for Dynamic Planners" in Proceedings of the Ninth National Conference on Artificial Intelligence, 744-749.)

Barto, A.G.; Bradtke, S.J.; and Singh, S.P. 1993. Learning to Act Using Real-Time Dynamic Programming. University of Massachusetts CMPSCI Technical Report 93-02. To appear in the *AI Journal.*

Chrisman, L., and Simmons, R. 1991. Sensible Planning: Focusing Perceptual Attention. In Proceedings of the Ninth National Conference on Artificial Intelligence, 756-761.

Dean, T.; Kaelbling, L.P.; Kirman, J.; and Nicholson, A. 1993. Planning with Deadlines in Stochastic Domains. In Proceedings of the Eleventh National Conference on Artificial Intelligence, 574-579.

Hendler, J., and Kinny, D. 1992. Empirical Experiments in Selective Sensing with Non-Zero-Cost-Sensors. In *Working Notes for the AAAI Spring Symposium on Control of Selective Perception*, 70-74.

Howard, R.A. 1960. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, MA.

Koenig, S. 1992. Optimal Probabilistic and Decision-Theoretic Planning using Markovian Decision Theory. UC Berkeley Computer Science technical report no. 685.

Langley, P., Iba, W., and Shrager, J. 1994. Reactive and Automatic Behavior in Plan Execution. To appear in *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems.*

Sutton, R.S. 1990. Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. In *Proceedings of the Seventh International Conference on Machine Learning*, 216-224.

Tan, M. 1991. Learning a Cost-Sensitive Internal Representation for Reinforcement Learning. In *Proceedings of the Eighth International Workshop on Machine Learning*, 358-362.

Whitehead, S.D., and Ballard, D.H. 1991. Learning to Perceive and Act by Trial and Error. *Machine Learning* 7:45-83.

DISTRIBUTION LIST

addresses	number of copies
WAYNE A. BOSCO	5
AFRL/IFTB	
525 BROOKS RD	
ROME NY 13441-4505	
UNIVERSITY OF MASSACHUSETTS	5
COMPUTER SCIENCE DEPT	
BOX 34610	
AMHERST MA 01003-4610	
AFRL/IFOIL	1
TECHNICAL LIBRARY	
26 ELECTRONIC PKY	
ROME NY 13441-4514	
ATTENTION: DTIC-OCC	2
DEFENSE TECHNICAL INFO CENTER	
8725 JOHN J. KINGMAN ROAD, STE 0944	
FT. BELVOIR, VA 22060-6218	
DEFENSE ADVANCED RESEARCH	1
PROJECTS AGENCY	
3701 NORTH FAIRFAX DRIVE	
ARLINGTON VA 22203-1714	
DR JAMES ALLEN	1
COMPUTER SCIENCE DEPT/BLDG RM 732	
UNIV OF ROCHESTER	
WILSON BLVD	
ROCHESTER NY 14627	
DR MARIE A. BIENKOWSKI	1
SRI INTERNATIONAL	
333 RAVENSWOOD AVE/EK 337	
MENLO PRK CA 94025	
DR MARK S. BODDY	1
HONEYWELL SYSTEMS & RSCH CENTER	
3660 TECHNOLOGY DRIVE	
MINNEAPOLIS MN 55418	

DR MARK BURSTEIN BBN SYSTEMS & TECHNOLOGIES 10 MOULTON STREET CAMBRIDGE MA 02138	1
DR THOMAS L. DEAN BROWN UNIVERSITY DEPT OF COMPUTER SCIENCE P.O. BOX 1910 PROVIDENCE RI 02912	1
DR PAUL R. COHEN UNIV OF MASSACHUSETTS COINS DEPT LEDERLE GRC AMHERST MA 01003	1
DR JON DOYLE LABORATORY FOR COMPUTER SCIENCE MASS INSTITUTE OF TECHNOLOGY 545 TECHNOLOGY SQUARE CAMBRIDGE MA 02139	1
DR MICHAEL FEHLING STANFORD UNIVERSITY ENGINEERING ECO SYSTEMS STANFORD CA 94305	1
RICK HAYES-ROTH CIMFLEX-TEKNOWLEDGE 1810 EMBARCADERO RD PALO ALTO CA 94303	1
DR JIM HENDLER UNIV OF MARYLAND DEPT OF COMPUTER SCIENCE COLLEGE PARK MD 20742	1
MR. MARK A. HOFFMAN ISX CORPORATION 1165 NORTHCHASE PARKWAY MARIETTA GA 30067	1
DR RON LARSEN NAVAL CMD, CONTROL & OCEAN SUR CTR RESEARCH, DEVELOP, TEST & EVAL DIV CODE 444 SAN DIEGO CA 92152-5000	1

DR. ALAN MEYROWITZ NAVAL RESEARCH LABORATORY/CODE 5510 4555 OVERLOOK AVE WASH DC 20375	1
ALICE MULVEHILL BBN 10 MOULTON STREET CAMBRIDGE MA 02238	1
DR DREW MCDERMOTT YALE COMPUTER SCIENCE DEPT P.O. BOX 2158, YALE STATION 51 PROPSPECT STREET NEW HAVEN CT 06520	1
DR DOUGLAS SMITH KESTREL INSTITUTE 3260 HILLVIEW AVE PALO ALTO CA 94304	1
DR. AUSTIN TATE AI APPLICATIONS INSTITUTE UNIV OF EDINBURGH 80 SOUTH BRIDGE EDINBURGH EH1 1HN - SCOTLAND	1
DIRECTOR DARPA/ITO 3701 N. FAIRFAX DR., 7TH FL ARLINGTON VA 22209-1714	1
DR STEPHEN F. SMITH ROBOTICS INSTITUTE/CMU SCHENLEY PRK PITTSBURGH PA 15213	1
DR JONATHAN P. STILLMAN GENERAL ELECTRIC CRD 1 RIVER RD, RM K1-5C31A P. O. BOX 8 SCHENECTADY NY 12345	1
DR EDWARD C.T. WALKER BBN SYSTEMS & TECHNOLOGIES 10 MOULTON STREET CAMBRIDGE MA 02138	1

DR BILL SWARTOUT 1
USC/ISI
4676 ADMIRALTY WAY
MARINA DEL RAY CA 90292

DR MATTHEW L. GINSBERG 1
CIRL, 1269
UNIVERSITY OF OREGON
EUGENE OR 97403

DR LESLIE PACK KAELBLING 1
COMPUTER SCIENCE DEPT
BROWN UNIVERSITY
PROVIDENCE RI 02912

DR SUBBARAO KAMBHAMPATI 1
DEPT OF COMPUTER SCIENCE
ARIZONA STATE UNIVERSITY
TEMPE AZ 85287-5406

DR MARTHA E POLLACK 1
DEPT OF COMPUTER SCIENCE
UNIVERSITY OF PITTSBURGH
PITTSBURGH PA 15260

DR MANUELA VELOSO 1
CARNEGIE MELLON UNIVERSITY
SCHOOL OF COMPUTER SCIENCE
PITTSBURGH PA 15213-3891

DR DAN WELD 1
DEPT OF COMPUTER SCIENCE & ENG
MAIL STOP FR-35
UNIVERSITY OF WASHINGTON
SEATTLE WA 98195

DIRECTOR 1
ARPA/ISO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

DR GEORGE FERGUSON 1
UNIVERSITY OF ROCHESTER
COMPUTER STUDIES BLDG, RM 732
WILSON BLVD
ROCHESTER NY 14627

DR STEVE HANKS DEPT OF COMPUTER SCIENCE & ENG'G UNIVERSITY OF WASHINGTON SEATTLE WA 98195	1
DR ADNAN DARWICHE INFORMATION & DECISION SCIENCES ROCKWELL INT'L SCIENCE CENTER 1049 CAMINO DOS RIOS THOUSAND OAKS CA 91360	1
DR. MAREK RUSINKIEWICZ MICROELECTRONCS & COMPUTER TECH 3500 WEST BALCONES CENTER DRIVE AUSTIN, TX 78759-6509	1
MAJOR DOUGLAS DYER/ISO DEFENSE ADVANCED PROJECT AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON, VA 22203-1714	1
DR. STEVE LITTLE MAYA DESIGN GROUP 2100 WHARTON STREET S&E 702 PITTSBURGH, PA 15203-1944	1
NEAL GLASSMAN AFOSR 110 DUNCAN AVENUE BOLLING AFB, WASHINGTON, D.C. 29332	1

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.